Joana da Silva Tavares

Bachelor in Computer Science

Secure Abstractions for Trusted Cloud Computation

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Bernardo Luis Silva Ferreira, Researcher,
NOVA University of Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça, Associate Professor,
NOVA University of Lisbon

Examination Committee

Chairperson: António Maria L.C. Alarcão Ravara
Raporteur: Alysson Neves Bessani
Member: Bernardo Luis Silva Ferreira

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Secure Abstractions for Trusted Cloud Computation

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

First I would like to thank my thesis advisor, Professor Bernardo Ferreira, and co-adviser, Professor Nuno Preguiça, for their guidance and support throughout the whole process of this thesis.

Above all, I would like to thank my sister, Zenaide Tavares, for being my biggest supporter and for having the biggest patience in the world. Lastly, I would like to thank my friends who accompanied me throughout these five years, with whom I shared good and some bad moments, and my colleagues for their camaraderie and shenanigans.

I am also thankful for the support from FCT/MCTES, through the strategic project NOVA LINCS (UID/CEC/04516/2013) and project HADES (PTDC/CCI-INF/31698/2017), and from the LightKone project (H2020 grant agreement ID 732505).

The bags under my eyes are Prada.

Abstract

Cloud computing is adopted by most organizations due to its characteristics, namely offering on-demand resources and services that can quickly be provisioned with minimal management effort and maintenance expenses for its users. However it still suffers from security incidents which have lead to many data security concerns and reluctance in further adherence. With the advent of these incidents, cryptographic technologies such as homomorphic and searchable encryption schemes were leveraged to provide solutions that mitigated data security concerns.

The goal of this thesis is to provide a set of secure abstractions to serve as a tool for programmers to develop their own distributed applications. Furthermore, these abstractions can also be used to support trusted cloud computations in the context of NoSQL data stores. For this purpose we leveraged conflict-free replicated data types (CRDTs) as they provide a mechanism to ensure data consistency when replicated that has no need for synchronization, which aligns well with the distributed and replicated nature of the cloud, and the aforementioned cryptographic technologies to comply with the security requirements. The main challenge of this thesis consisted in combining the cryptographic technologies with the CRDTs in such way that it was possible to support all of the data structures functionalities over ciphertext while striving to attain the best security and performance possible.

To evaluate our abstractions we conducted an experiment to compare each secure abstraction with their non secure counterpart performance wise. Additionally, we also analysed the security level provided by each of the structures in light of the cryptographic scheme used to support it. The results of our experiment shows that our abstractions provide the intended data security with an acceptable performance overhead, showing that it has potential to be used to build solutions for trusted cloud computation.

Keywords: Trusted cloud computing, Homomorphic Encryption, Searchable Encryption, Data Abstractions, CRDTs

Resumo

O paradigma de computação na nuvem é adoptado pela maioria das organizações devido às suas características, isto é por providenciar recursos e serviços *on-demand* com custos de gestão e manutenção mínimos para os utilizadores. Todavia, existem vários incidentes relativos à segurança dos dados alojados na nuvem. Como tal, surgiram preocupações quanto ao grau de segurança da nuvem o que por sua vez despoletou relutância a uma maior adesão aos seus serviços. No seguimento dos incidentes, surgiram soluções que se baseiam em tecnologias criptográficas, como cifras homomórficas e pesquisáveis, e que permitem mitigar os problemas de segurança.

O objectivo desta tese consiste em disponibilizar um conjunto de abstracções seguras que permitem desenvolver aplicações distribuidas e seguras e que possibilitem a computação confiável na nuvem no contexto de bases de dados NoSQL. Para o efeito, tomámos partido de tipos abstractos de dados replicados e livres de conflito (CRDTs) dado que estes fornecem um mecanismo que garante a consistência de dados quando replicados, sem recorrer a sincronização, que se conjuga bem em ambientes distribuidos. Para cumprir os requisitos de segurança, tomámos partido das tecnologias criptográficas previamente referidas. O principal desafio desta tese consistiu em combinar estas tecnologias com os CRDTs de forma a ser possível suportar todas as suas funcionalidades sobre texto cifrado e de forma a garantir o melhor grau de segurança e desempenho possível.

Para efeitos de avaliação experimental, comparamos o desempenho das abstracções seguras com as suas respectivas abstrações não seguras. Adicionalmente, também efectuamos uma análise do nível de segurança providênciado pelas estruturas desenvolvidas em função das tecnologias criptográficas subjacentes às mesmas. Os resultados obtidos demonstram que as abstracções cumprem com os requisitos de segurança pretendidos com uma penelização de desempenho aceitável, o que revela o seu potencial para serem utilizadas em soluções para computação confiável na cloud.

Palavras-chave: Computação confiável na nuvem, Esquemas de Cifras Homomórficas, Esquemas de Cifras Pesquisáveis, Abstrações de Dados, CRDTs, SGX

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The cloud computing paradigm has received a substantial amount of attention in the past decade. It emerged as a computational paradigm for on-demand network access to a shared pool of computing resources such as network, servers, storage, applications, and services that can be quickly provisioned with minimal management effort and maintenance expenses [42].

Due to its characteristics it sparked a shift of computer processing, storage, and software provisioning away from the desktop and local servers into the next generation of data centers which are hosted by large infrastructure companies such as Amazon, Google, Yahoo, Microsoft, or Sun. In fact, many applications have been created in or migrated to cloud environments over the last few years [32].

Parallel to the increase in its popularity, the concern for the security and privacy issues inherent to cloud computing also grew. From the consumer's perspective, the concern stems from the fact that they have little or no control over their data or the applications and infrastructure involved since the cloud services are delivered by a cloud service provider and accessed via the Internet. As a consequence, many companies which deal with more sensitive data became skeptical and refrained further adoption of cloud solutions. From the perspective of cloud service providers, the security concern derives from their vulnerabilities to attacks and failure in equipment, software and controls [11].

Amongst its many security issues, data security which refers to the protection of data privacy and integrity throughout its life cycle is the most pressing one [19]. Its a particularly challenging issue because besides data owners not having full control of the data life cycle, data locality and jurisdictional issues can prevent cloud service providers from protecting personal information.

The most common solution to mitigate data security issues is through encryption and key management [11, 18, 50]. For effective encryption, the encryption algorithm and key strength must also be taken into consideration.

Intuitively, the encryption solution would consist in the owner encrypting the data, with conventional encryption before sending it to the cloud and when needed, he would decrypt the data after retrieving it from the cloud. This solution allows to safely store the data in the cloud, yet it becomes unsuitable when computations or queries must be performed over the data, which typically is the case, as it places the burden of such operations on the client-side and it wastes the cloud's computation resources that could be put to use to perform the operations more efficiently.

1.2 Motivation and Problem Statement

Encryption schemes that enable operations over ciphertext, such as homomorphic [13, 48] and searchable [16] encryption schemes, have been leveraged to benefit from cloud computing while still preserving data security. Furthermore, these schemes enabled secure computations to be performed in the cloud as they ensure the correctness as well as the privacy of such operations despite the fact that they are executed in untrusted cloud servers. Considerable amount of research was done in order to provide solutions for secure computing, which could then be leveraged for cloud computing based on the aforementioned encryption schemes like encrypted relational databases such as CryptDB [51] and Cipherbase [6]. These solutions meet the security requirements with acceptable overheads to the overall performance of the system.

In the past years with the advance of Web technologies, the integration and proliferation of sensors in multiple devices and mobile devices that are connected to the Internet and the increase of volume of data generated by businesses, unprecedented amounts of data for which storage and processing capabilities are need have been produced [32]. As traditional relational databases became inadequate to deal with such amounts of data, a new type of data store emerged — Not Only SQL (NoSQL). These are highly available, scalable and fault-tolerant and therefore are very suitable to be used within the cloud context. High availability is mostly provided through replication mechanisms [22], which raises the issue of maintaining the replicated data consistent and solving conflicts of concurrent manipulation of different copies of the data. Conflict free replicated data types (CRDT) [64] are data abstractions that address this issue by providing a built-in conflict resolution mechanism that doesn't require synchronization. Riak [54] is an example of a NoSQL data store that provides such structures.

However when compared to the relational database systems, NoSQL data stores offer a smaller set of security mechanisms [32, 43]. To improve the offer and richness of security mechanisms in these stores, there has been research efforts to provide solutions for this issue such as the framework to preserve privacy presented by Macedo et. al. [40]. The proposed framework enables data processing over multiple cryptographic techniques and

is composable with many NoSQL engines. This solution can be configured for different security requirements and is able to meet them with an acceptable overhead to the overall performance of the system.

In summary, when it comes to the development of secure databases, either relational or NoSQL, through the use of cryptographic techniques there is a trade off between performance and security. Providing strong security might lead to a system that is not available nor scalable. Conversely, providing low or no security in order to match performance requirements can lead to data security breaches.

In this thesis the problem we focus on is how can we security enhance NoSQL databases while keeping to a minimum the impact on the performance of the database itself and, most importantly, without restricting the ability to perform computations over the stored data.

In this thesis we focus on the problem of how can we enhance the security of NoSQL databases while keeping to a minimum the impact on the performance of the database itself and, most importantly, without restricting the ability to perform computations over the stored data.

1.3 Objectives and Contributions

This thesis aimed to develop a set of secure abstractions to support secure computations in the context of NoSQL data stores hosted by cloud services and to serve as a tool for programmers to develop their own distributed applications. In the scope of this thesis, the term security refers to the ability of maintaining the data involved in the computations confidentiality and privacy protected.

To achieve this we leveraged CRDTs as their characteristics make them suitable to be used in distributed systems or applications and encryption schemes, such as homomorphic encryption, to enable operations over the abstractions with security guarantees. The secure abstractions include typical data-structures that form the backbone of today's more complex distributed and cloud applications. The main challenge of this thesis consisted in combining the encryption schemes with the CRDTs in such way that it was possible to support all of the data structures functionalities over ciphertext while striving to attain the best security and performance possible.

In our threat model we consider a cloud administrator adversary which is of the type Honest but Curious (HbC) [19]. This administrator has access to the cloud provider's infrastructure and software platforms which consequently provides him access to customer data. The HbC administrator main goal is to gather as much information about the computations and data as possible in order to infer their nature.

To understand the impact the security enhanced CRDTs, we performed micro benchmarks to compare performance wise each secure abstraction with their non secure counterpart. Additionally, we also analysed the security level provided by each of the structures in light of the cryptographic scheme used to support it.

In essence, we made the following contributions:

1. Design and implementation of a library of CRDTs;
2. Design and implementation of a library of secure CRDTs, which is a variant of the aforementioned library, supported by cryptographic primitives with property-preserving and homomorphic properties;
3. Extensively evaluate the developed contributions regarding both libraries.

1.4 Document Structure

This document is structured in six chapters, including the current one. Chapter 2 contains the related work of this thesis, which consist in hardware-backed trusted computing solutions, computing on encrypted data and data stores and models. Along the chapter there are subsections dedicated to discuss in some capacity each of the addressed topics, and in at the end of it there is a summary of these discussions. Chapter 3 provides an overview of our solution, addressing its system and threat model, and Chapter 4 describes the technical implementation aspects of the solution. On Chapter 5, the description of the experimental evaluations we performed is given along with their results and critical analysis. Finally, Chapter 6 presents our conclusion, contributions and future work.

Chapter 2

Related Work

This chapter is structured in 3 sections. In the first two sections, we will discuss different solutions which provide privacy, availability and integrity of outsourced data in the cloud. Section 2.1 discusses trusted computing solutions which are based in hardware, section 2.2 addresses solutions based on cryptography. In the last section (2.3) NoSQL stores are analysed as a solution for distributed storage systems.

2.1 Hardware-backed Trusted Computing

Trusted computing refers to the expectation of technologies to not compromise their claimed security properties. These properties are enforced through hardware/software solutions. The hardware approach to trusted computing is referred to as Trusted Hardware [67].

In this section we will discuss the state of the art of implementations of trusted hardware, focusing on the most relevant solutions for trusted execution environments (TEE) solutions — Intel's SGX [20]) and ARM's Trustzone[8] — while still addressing Trusted Platform Modules (TPM) [68] as it is at the core of trusted hardware.

2.1.1 Trusted Execution Environments (TEE)

A TEE is a secure, integrity-protected execution environment, consisting of processing, memory, and storage capabilities that is isolated from the normal execution environment [10]. In this context, isolation refers to the ability to run security-critical code outside the control of the normal execution environment which is not trusted.

A TEE architecture may support one or more TEE instances. Architectures based on dedicated security chips and processor modes (e.g. TPMs [68] and ARM's TrustZone [58]), often have only one instance available. In this scenario, the same instance typically

allows execution of multiple trusted applications. Additionally, an interface that enables communication with trusted applications and invocation of cryptographic operations within the TEE is provided to applications from the normal execution environment.

Architectures based on virtualization (e.g. Terra [29], TCCP [56]) and emerging processor architectures (e.g. Intel SGX [20]) allow the normal execution environment application to create or activate a TEE instance when needed through an application programming interface (API). This API allows the application to execute trusted applications and to read and write data to and from them.

2.1.2 Trusted Platform Modules

A TPM is a hardware module incorporated and deployed in a motherboard, smart card or processor that provides the resources needed for trusted computing. As shown in Figure 2.1, the TPM has several components.

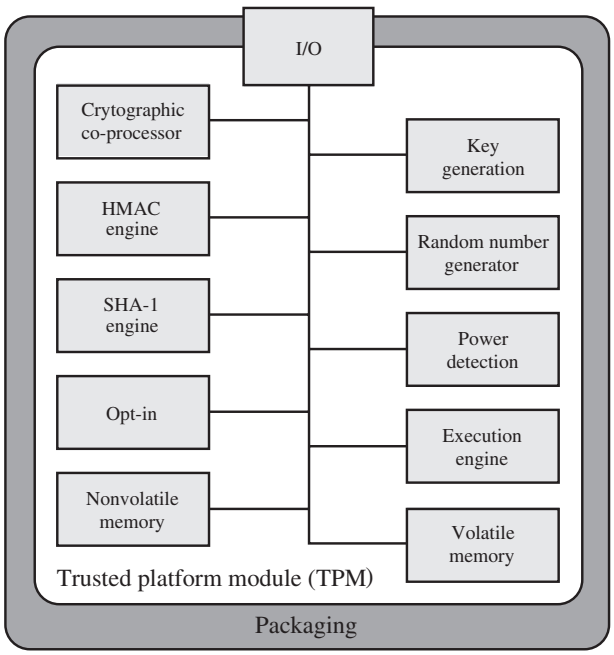

Figure 2.1: TPM architecture (taken from [67]).

Through secret sharing and cooperation with other hardware and software components, the TPM provisions three services: authenticated boot, certification, and encryption.

The authenticated boot service allows to verify that the boot of the entire operating system is made in well defined stages at which only approved versions of the modules of the OS are loaded. This could be done by verifying a digital signature associated with each module. Additionally, a tamper-proof log of the loading process is kept by the TPM that later can be consulted. Tampering is detected using cryptographic hash functions. It is also possible to configure the TPM to include additional hardware, application and

utility software in its trust computing base (TCB) given some restrictions to prevent threats. This service can be used to guarantee that the machine which hosts the TPM is in a well defined and trusted state after booting.

Through the certification service TPMs support attestation. Attestation is a security mechanism that allows a third party to verify that the contents of a secure container, provided by trusted hardware, is as expected. Typically the proof consists in a piece of attestation data signed by the trusted hardware, producing an attestation signature.

In the TPM the attestation data is the log of the loading process produced by the authenticated boot service. The data is then signed, using a private key exclusive to attestation, and a digital signature certificate is produced. Upon receiving the certificate, the third party only needs to verify the signature using the TPM's public key. The third party can trust this proof because the TPM is considered trustworthy and only the TPM possesses its private key. To prevent message replay attacks, a nonce is sent along with the certification request issued by the third party.

The encryption service enables the encryption of data that can only be decrypted by a machine with a certain configuration. For this purpose, a secret key is generated from a never exposed master secret key owned by the TPM and which is unique to each machine. The key generation algorithm used by the TPM generates keys for every possible machine configuration and then binds each key to the corresponding configuration. Thus, the configuration of the machine at the time of encryption must match the configuration when decryption occurs for it to succeed. Applications can leverage this service to encrypt data while running in a machine whose configuration is trusted and safe by wrapping the key used for data encryption with a key provided by the TPM. When decryption of the data is required, the wrapped key is submitted to the TPM to be unwrapped. The TPM verifies if the current system configuration allows access for the key required for unwrapping and if the application that issued the request is authorized to access it. If so, the original key is returned, and the application can decrypt its data.

TPMs can provide more minimal TEEs. This can be achieved through the certification service as after the validation of the correctness and integrity of the software stack that runs on the TPM enabled machine, an trusted execution environment with processing, memory, and storage capabilities has been secured. This TEE is considered more minimal as there isn't complete isolation of the environment.

Santos et. al. [57] leveraged TPMs to provide a policy-sealed data abstraction. This abstraction allows customer data to be bound to cloud nodes whose configuration is specified by a customer-defined policy. A configuration consists in a set of attributes that express features that refer to the node's software or hardware. A policy expresses a logical condition over the attributes that must be supported by the provider.

2.1.2.1 Security Overview

TPM's can suffer 3 types of attacks: software, reset and timing attacks. Physical attacks to the TPM are excluded from its threat model [66].

Software attacks target the attestation service. The main issue is that the configuration that is certified may not be the one that is currently in the system as the measurements are taken upon loading of the system. Given an application software that has been configured in the TPM, an adversary can still exploit the difference between the instant when the software is measured and when it is actually used to induce run-time vulnerabilities [14]. These attacks are classified as time-of-check time-of-use (TOCTOU). A possible solution is to have the system where the TPM lives monitor the state of memory of the application it is attesting to. Upon change, either the changes are incorporated in the measurement or they are reported to the operating system.

Reset attacks [66] are characterized by instructing the TPM to perform a hardware reset, even if the rest of the system is not instructed to restart, which will make it return to its uninitialized state. Thereafter, a malicious configuration can be setup on the TPM as follows. Before instructing the TPM to restart, the attacker obtains copies of the digital signatures of the trusted configuration which were fed to the authenticated boot service. Then the attacker swaps the operating system for a malicious one by exchanging the machine's hard drive, for example. Upon booting, the hardware reset must be issued to the TPM and the malicious operating system must feed the TPM with the previously obtained signatures of the trusted configuration. At the end of the attack, the malicious operating system will be able to perform whichever operations were bound to the trusted configuration.

To carry out the previously described attack, its is necessary to record the trusted boot process. This is achieved through timing attacks, more specifically by snooping the unsecure communication channel between the TPM and CPU at instances where it is known sensitive information is being exchanged (e.g. startup) [38]. This attack is easily circumvented by the creation of an encrypted channel. According to the TPM specification [68], its possible to create one although there is still doubt if the processor will be able to secure one with the TPM.

2.1.3 Intel SGX

Intel's Software Guard Extensions (SGX) is a set of x86 instructions and memory access changes to the Intel architecture [41, 47]. SGX is leveraged by applications to ensure confidentiality and integrity guarantees, even when the privileged software on the computer where security sensitive computations are being performed are not trusted.

These guarantees are provided through the use of a TEE referred to as an enclave. An enclave is a protected physical memory region isolated from the remainder code of the system, including the operating system and hypervisor. Each enclave contains the code and the data required to perform the security-sensitive computations which is provided

by the user. Therefore, the trusted computing base (TCB) of Intel's SGX is composed by the processor and code chosen by the user [9, 60].

Isolation is achieved by storing the enclave's code and data, which corresponds to the security sensitive computation's data and code, in the Processor Reserved Memory (PRM) which is memory that can't be directly accessed by other software. This memory is encrypted and integrity protected at all times, except when moved to the processor.

Any other enclave specific information is stored in the Enclave Page Cache (EPC), which is a subset of the PRM and it is managed by the system software (e.g. hypervisor). This implies that although the system software is not trusted, it is still relied upon to manage an enclave. Multiple enclaves are supported by having the EPC fragmented in pages that can be assigned to different enclaves [20]. However, as the enclave's memory has an imposed limit of either 64MB or 128MB, the number of active enclave's, i.e. in memory, is limited [35].

To leverage Intel's SGX without compromising the offered security guarantees and injecting vulnerabilities, the submitted code must comply with the following guidelines:

- It must be divided in two logical components: trusted and untrusted. The trusted component defines the security sensitive computations. It represents the enclave. The untrusted component defines the rest of required operations;
- The trusted component should be as small as possible. It should only operate over secret data;
- The trusted component code may access unprotected memory and call functions in the untrusted component although there should be a minimal dependency on the untrusted component.

An enclave's life cycle can be described as a finite state machine as shown in Figure 2.2. Table 2.1 describes the most relevant instructions which coincide with state transitions.

Table 2.1: Main instructions of an enclave's life cycle.

Instruction	**Description**
ECREATE	Creates an enclave by setting up the it's internal data structures, which are allocated within the PRM, with information retrieved from unsafe memory regions.
EADD	Loads the initial code and data into the enclave from unsafe memory regions.
EEXTEND	Updates the enclave's measurement which is used in the software attestation process.
EINIT	Marks the enclave's internal data structures as initialized which enables its for execution.
EREMOVE	Terminates an enclave by deallocating all of it's memory.

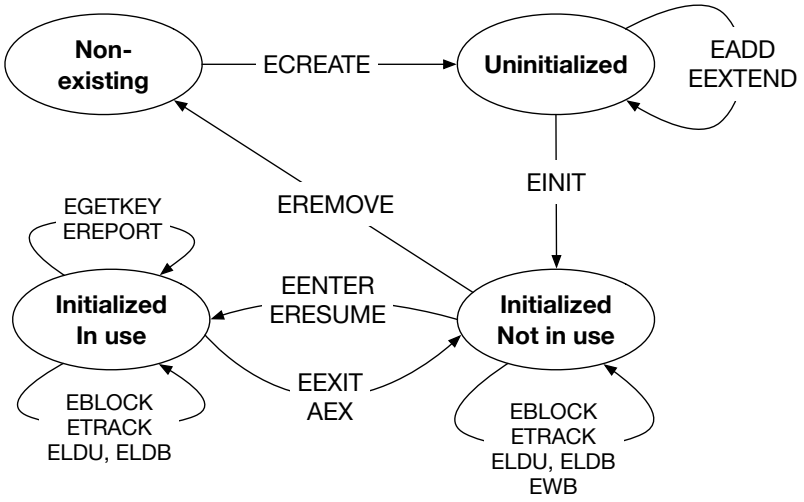

Figure 2.2: Intel SGX's enclave life cycle (taken from [20]).

SGX, similarly to TPMs, also provides attestation. In Intel's SGX, the proof consists in a cryptographic signature that certifies a measurement of the secure container contents which was computed at the creation of the enclave. The signing of the measurement is performed by a privileged enclave, the Quoting Enclave, that can access the required attestation key.

Intel's SGX originally was designed for securing small services but as cloud computing has become a big trend, it has been leveraged to provide security guarantees to applications deployed in the cloud. One example is VC3 [60], proposed by Schuster et. al., which allows users to run distributed MapReduce computations in the cloud while keeping the data and code private. Another example is the solution developed by Ohrimenko et. al. [47], which consists in a set of data-oblivious machine learning algorithms for collaborative data analytics that maintains individual datasets private.

2.1.3.1 Security Overview

The threat model of SGX includes the following category of attacks [20]:

- Privileged software attacks: most Intel processors support hyper-threading which means that the CPUs, execution units and caches on a single core are shared by two logic processors (LP), with each having their own state. As SGX doesn't prevent hyper-threading, malicious system software may leverage this feature to carry out an attack. Specifically, it can schedule a thread that is executing the code of an enclave on an LP that shares its core with another LP that is executing a snooping thread. This allows for the snooping thread to learn the instructions executed by the enclave and its memory access patterns;

- Memory mapping attacks: SGX is vulnerable to passive address translation attacks, due to using the address translation process provided by the system software, which

can be leveraged to learn the enclave's memory access patterns at page granularity;

- Software side-channel attacks: SGX doesn't provide protection against software side-channel attacks that rely on performance counters which can leak information about the enclave.

Categories such as physical attacks to the CPU chip, side-channel attacks and cache timing attacks do not feature in its threat model. Nevertheless, researchers are still actively proposing new solutions for the last two categories as new forms of attacks are discovered. This could be due to the popularity of SGX for cloud applications.

In [65], the authors address page-fault-based side channel attacks which allows the malicious OS to gain complete control over the execution of SGX programs by stopping the program, unmapping the target's memory pages and resuming execution. To mitigate this attack, the Transaction Synchronization Extensions (TSX) of Intel processors are used to abort ongoing transactions upon unexpected exceptions or interrupts. The authors of [61] address memory corruption attacks with a new Address Space Layout Randomization (ASLR) scheme built on top of SGX that secretly bootstraps the memory space layout with a finer-grained randomization.

2.1.4 ARM Trustzone

ARM's TrustZone is the security extension made to ARM's System-On-Chip (SoC) that covers the processor, memory, and peripherals [58]. It provides a hardware-level isolation between two execution domains: the normal world and secure world. The secure world hosts a secure container while the normal world runs an untrusted software stack [15].

These domains have independent memory address spaces and different operating systems and privileges: while code running in the normal world cannot access the secure world address space, code running in the secure world can access the normal world address space.

On boot, the processor initializes in the secure world and then sets up the necessary environment before switching to the normal world. As such, the TrustZone's TCB consists in the boot loader, the processor and the software that will run in the trusted world. When required, the execution can switch into the secure world using the secure-monitor call (SMC) instruction. The SMC instruction generates a software interruption that is trapped by the chosen secure monitor implementation. Additionally, interrupt requests can be configured to be secure or non-secure and are handled accordingly. Therefore, secure interrupts will only be handled by the secure world. This mechanism alongside the supervision of the CPU state through the NS bit, which indicates the current world of the processor, and the partitioning of the memory address space into secure and non-secure regions guarantees the isolation between worlds.

To use the secure services provided in the secure world, communication between worlds is provided through shared memory. This memory is physically allocated in the

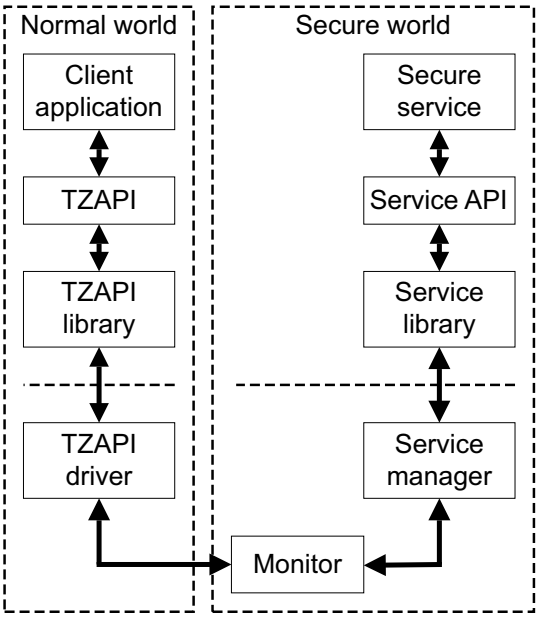

Figure 2.3: Example of a system architecture using the TrustZone API (taken and adapted from [8]).

normal world's memory region. Processes within the secure world can still gain access to it, due to having higher privilege, by mapping a normal world address to its page table [37]. The shared memory is used to specify which services were required and the necessary input arguments.

Although ARM's TrustZone can also be leveraged for cloud applications, it has been more exploited in mobile devices as its architecture is present in many of them for over a decade. Examples of such is the API of Ekberg et. al. [23] that provisions trusted applications and secrets to the device, authorizes trusted applications to access the provisioned secrets and device keys, and control which normal world application can execute trusted applications and the system proposed by Santos et. al. [58] that protects confidentiality and integrity of .NET mobile applications from OS security breaches. Darkroom [15] is an example of the use of TrustZone for cloud applications. It enables users to securely process image data in the cloud while preventing the exposure of sensitive data to the operating system.

2.1.4.1 Security Overview

TrustZone can be configured to provide countermeasures for different threats, therefore it doesn't possess a fixed threat model. To build a configuration that meets a threat model specification, security protocols must be built on top of TrustZone (e.g. attestation) and combined with parts of the system which are made secure through TrustZone's architecture.

Nonetheless, attacks to the TrustZone architecture are still studied and identified by

researchers. For an instance, Jang et. al. [37] addresses the unauthenticated access to resources in TrustZone which makes the channel used for communication between worlds vulnerable to an attacker with the normal world's kernel privilege. The attack can consist in creating a malicious process that continuously sends requests with crafted arguments to discover the vulnerabilities of the resources in TrustZone. Another possibility is a man-in-the-middle attack. To mitigate these attacks, the authors proposed a framework that allows the creation of a secure channel for cross worlds communication.

2.1.5 Critical Analysis

As there is no specification related to the TPMs minimum performance requirements, commodity TPMs are slow and inefficient due to the lack of incentive to use faster yet more expensive internal components. Consequently, the usage of TPMs is limited to use cases that don't require fast nor frequent operations.

In general, a drawback of TEEs is the lack of standardization regarding an API for developers to make use of which has hindered the development of real world applications which leverage them. Yet, the Global Platform association and the Trusted Computing Group have made standardization efforts as the interest and demand for greater levels of security in applications arises [23].

2.2 Computing on Encrypted Data

In the context of cloud environments, computation security may be divided into two classes — cloud storage security and cloud computing security [69]. Cloud storage security refers to ensuring the integrity and confidentiality of outsourced data stored at untrustworthy cloud servers while cloud computing security refers to verifying the correctness of the outsourced computation performed by untrustworthy cloud servers. In this section, we will discuss cryptographic schemes (2.2.1) and systems (2.2.2) that provide solutions for both classes.

2.2.1 Cryptographic Schemes

2.2.1.1 Homomorphic Encryption

Homomorphic encryption designates cryptographic schemes for which, given a fixed key, it is equivalent to perform operations, specifically addition and/or multiplication, on the plaintexts or on the corresponding ciphertexts [28]. In other words, these schemes allow a set of operations to be performed directly over encrypted data with no need for decryption while still obtaining the correct result.

These schemes enable the outsourcing of resource-intensive computing tasks to the cloud while maintaining confidentiality. Next, three forms of homomorphic encryption

will be presented — Fully, Partial and Somewhat Homomorphic Encryption — where the main difference among them is the set of operations allowed.

Somewhat Homomorphic Encryption Schemes that a allow a variable yet finite number of operations to performed over ciphertext are referred to as Somewhat Homomorphic Encryption (SHE or SWHE) schemes. The term *somewhat* is due to all schemes having the following property: as the number of operations performed increases, the noise factor increases exponentially and if a threshold p for this factor is exceeded, decryption of the ciphertext is no longer possible. An example of such scheme is the one presented by Boneh et. al. [13], which is a public key encryption scheme that allows addition and (very limited) multiplication over ciphertexts through quadratic formulas.

Fully Homomorphic Encryption Firstly introduced by Rivest et al. in 1978 [55], Fully Homomorphic Encryption (FHE) schemes allow an arbitrary number of operations to be performed over ciphertext. Only in 2009 a feasible scheme was presented by Craig Gentry [30]. It relied on a ideal lattice based SHE to which a boostrapping procedure was applied, resulting in a reduction of the noise factor that transformed the SHE into a FHE. Although it was an important achievement, this scheme is not usable in practice due to is big overhead. Although other schemes have been proposed since, in general, albeit ideal, FHE is impractical due to the overhead of the required computations that renders them unable to meet performance requirements of applications.

Partial Homomorphic Encryption Due to the inefficiency of FHE, researchers developed Partial Homomorphic Encryption (PHE) schemes which only allow a single operation over ciphertext. Typically, the operation is either addition or multiplication but never both. These schemes are much more efficient and practical as they rely in conventional cryptography such as public-key cryptography and modular arithmetic. An example of such scheme is Paillier's [48], which leveraged a trapdoor technique based on composite-degree residues to allow addition over ciphertexts. Another example is ElGamal's scheme [24], which is based on the difficulty of computing discrete logarithms over finite fields and allows multiplication over ciphertexts.

2.2.1.2 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) is a solution which addresses the problem of efficiently searching for keywords over remote encrypted data, where the underlying cryptography is symmetric. In this problem typically there is a user, which owns the data, and an untrusted server, where the encrypted data is stored [36]. The encryption of the data is made by the user, who may organize it in an arbitrary way and include additional data structures to allow for efficient access of relevant data [21]. With the rise of cloud storage solutions, which motivates storage outsourcing, this problem gained more attention and consequently so did searchable encryption schemes.

To be able to search within ciphertexts, determinism is required both for efficiency reasons and to actually match the queried keyword. However, determinism reveals information patterns to adversaries upon execution of certain operations (e.g. searches), which may conflict with the desirable security guarantees [25].

An example of an SSE scheme is [16] which allows to search over encrypted records. In this scheme for each record/keyword pair there is pseudo-random label. All labels and corresponding record identifiers, which are encrypted, are kept in a generic dictionary data structure in the server. Whenever a query is executed, the client computes a short key derived from the label associated with the queried keyword. Then, the server uses this short key to search the dictionary for the matching encrypted record identifiers. After retrieving the identifiers, the client decrypts them and obtains the result of the query. Besides searching, this scheme also allows to update the data set stored remotely.

Other examples of SSE schemes, that follow the approach of the aforementioned scheme but go beyond text search, are BISEN [26] and MuSE [27]. BISEN allows to search for documents matching a boolean expression with multiple keywords while MuSE allows cloud-backed applications to dynamically store, update, and search datasets containing multiple media formats.

2.2.2 Cryptographic Systems

2.2.2.1 CryptDB

CryptDB [51] is a middleware solution that provides confidentiality for applications that use database management systems (DBMS) by enabling a range of SQL queries to be made over encrypted data.

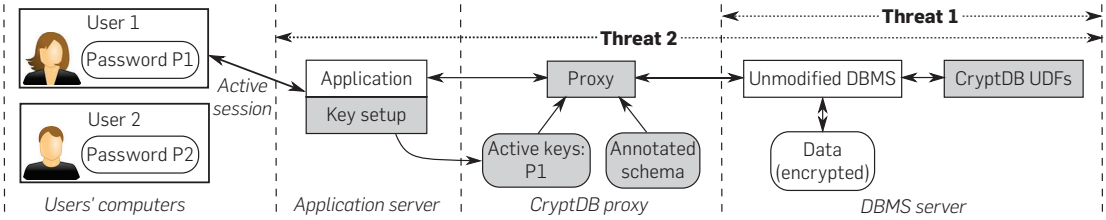

Figure 2.4: CryptDB's system architecture (taken from [51]).

It addresses two threats (Figure 2.4), specifically (1) an adversary who gains access to the DBMS server and tries to learn private data and (2) an adversary who gains complete control over the application and DBMS servers. In the first threat, CryptDB aims to prevent the adversary of learning private data and on the second threat it aims to protect the confidentiality of data owned by users which are logged-out of the application during an attack, providing no such guarantees for logged-in users.

In this solution, a trusted proxy intercepts issued queries and transforms them so that they may work over the encrypted data stored at the server while preserving the semantics of the original query. Conversely, upon receiving a query result, the proxy transforms (i.e.

decrypts) the result so that the application may process it. As such, the transformations include encryption decryption, change of query operators and obfuscation of names of columns and tables.

To mitigate information leakage due to some of the encryption schemes used (e.g. OPE), an onion encryption mechanism is provided. This mechanism consists in composing different layers of encryption, which allow for different operations to be performed, over a single data item in such a way that the weaker encryption scheme resides in the innermost layer and the stronger scheme resides in the outermost layer of the onion.

2.2.2.2 Cipherbase

Cipherbase [6] is a full-fledged SQL database system, that extends Microsoft's SQL Server, which allows for organizations to leverage the advantages of cloud computing platforms while providing data confidentiality by storing and processing encrypted data. It incorporates customized secure co-processors at the server side to distribute computations between trusted hardware, which resides in a Trusted Machine (TM), and untrusted hardware, which resides in an Untrusted Machine (UM). The secure co-processor in this system is a Field Programmable Gate Array (FPGA).

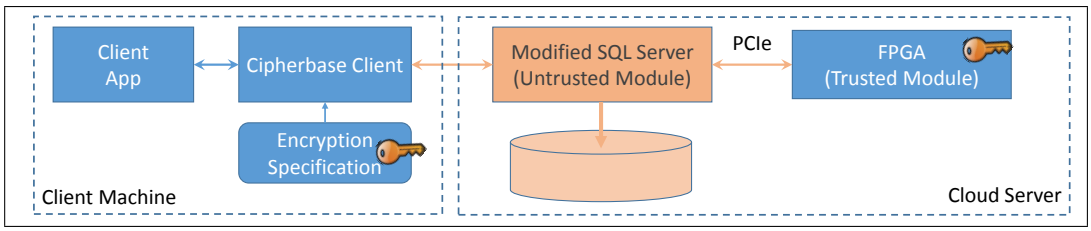

Figure 2.5: Cipherbase's system architecture (taken from [7]).

When a client executes a query that requires computation over encrypted columns, which are specified by the client, tuples are transferred from the UM to the TM. To obtain and return the query result to the UM, which is encrypted, the tuples are decrypted, processed and re-encrypted by the TM. To this extend, Cipherbase simulates fully homomorphic encryption. However, if the tuples are encrypted with homomorphic schemes their properties are used to perform the operations on the data in the UM.

2.2.2.3 Depsky

DepSky [12] is a middleware cloud storage solution that leverages different commercial clouds, forming a cloud-of-clouds, to provide a dependable storage system with availability, integrity and confidentiality guarantees. To do so, DepSky employs an byzantine fault-tolerant replication across the different clouds, a secret sharing scheme plus erasure codes and encryption to store data.

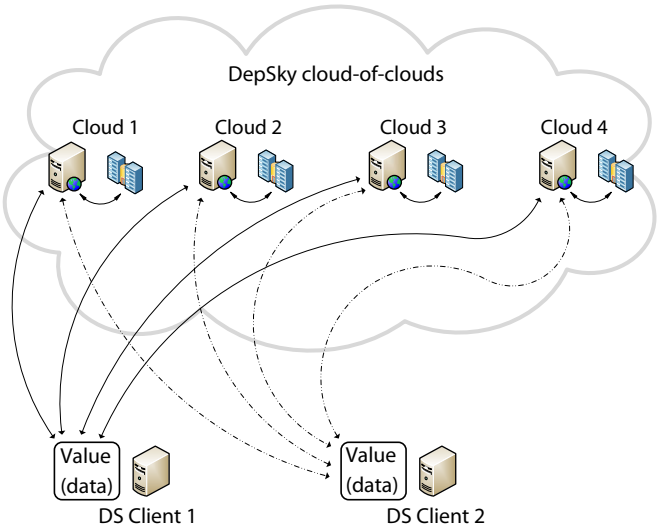

Figure 2.6: Depsky's system architecture (taken from [12]).

To store a data item in DepSky, first the item is encrypted with a random secret key. This key will be divided and distributed across the cloud servers with the secret sharing scheme, which consists in distributing shares of the secret to the servers. The main property of this scheme is that only a subset of all the shares that form the original secret is needed to recover it. After encryption, the data item is encoded and distributed among the servers.

The combination of these techniques ensures that no individual cloud is able to access the unencrypted item while allowing duly authorized clients to recover the subset of shares required to recover the original the item. However it adds an overhead of 50% the amount of storage spaced used in each cloud when compared with the original data size.

2.2.2.4 Cuttlefish

Cuttlefish [59] is a system that ensures the confidentiality of data stored in the cloud. It enables computations over encrypted data, leveraging both PHE and TEEs for this purpose, by transforming submitted queries into semantically equivalent ones.

Cuttlefish also provides a set of secure data abstractions (SDTs) with corresponding annotations. These abstractions capture restrictions on computations with respect to confidentiality which can significantly improve performance. The SDTs provided are sensitivity level, data range, decimal accuracy, uniqueness, tokenization, enumerated type and composite type. Table 2.2 describes the SDTs annotations.

Figure 2.7 ilustrates the main components of this system. To improve expressiveness and performance while reducing the amount and extent of re-encryption needed, the Cuttlefish compiler employs the following techniques:

- Expression rewriting: expressions that are not supported by Cuttlefish's encryption

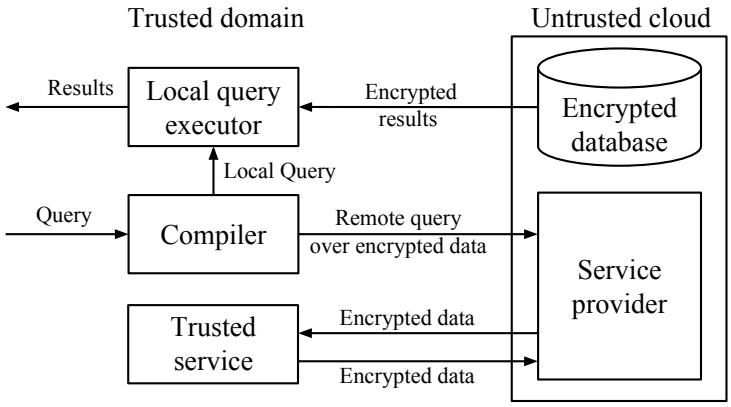

Figure 2.7: Cuttlefish's system architecture (taken from [59]).

schemes are replaced by equivalent and supported one's. The replacement query is also written in encryption-sensitive ways in order to reduce execution latency.

- Condition expansion: the compiler expands conditions to improve execution performance by eliminating expensive homomorphic encryption and/or re-encryption.
- Selective encryption: fields that do not contain sensitive information are kept in plaintext to reduce the overhead of operating over encrypted data.
- Efficient encryption: when one field is involved in multiple operations that can be supported by the same homomorphic encryption scheme, Cuttlefish selects the homomorphic encryption scheme that allows for a more efficient execution.

Table 2.2: Cuttlefish SDTs annotations (taken and adapted from [59]).

Annotation	**Description**
`+ | -`	Positive or negative numeric values
`range(x-y)`	Values from x to y
`accuracy(x)`	x decimal points preserved
`unique`	No duplicate values
`delimiter(char)`	Tokens separated by `char`
`enum(v1, v2, ...)`	Enumerated values
`composite`	Composite values

When a user submits a query, first the compiler transforms it into a remote and local query. The remote query, which operates over encrypted data, is issued to the untrusted cloud. Upon the remote query completion, the encrypted results are returned to the local query executer. These results are used as the local query executor input which decrypts them and performs any remaining computations on the plaintext data before returning the final results to the user.

When a query cannot be executed in the cloud without leaking information, the trusted re-encryption service that as access to the decryption keys is used. If need be, some transformations are also performed on the data before re-encryption with a different scheme occurs. Afterwards, the re-encrypted data is returned to the untrusted cloud so that the remote query can be completed. This service can be implemented using hardware at the client-side or using trusted hardware, namelly Intel SGX, in the cloud.

2.2.3 Critical Analysis

Although cryptographic schemes enable operations to be done over ciphertext, these schemes leak information about the performed operations (as is the case of SSE) and they either have a big performance overhead which renders them unusable or only allow limited operations to be performed (homomorphic encryption).

In the case of SSE, the information that is leaked mainly comprises access and search patterns [25, 39]. Access patterns reveal which documents contain the queried keyword, even though the adversary never learns the keyword in plaintext. This can be enough to disclose sensitive information about the original data, given some prior knowledge about it. This type of information leakage has been proved impossible to stop by Muhammad Naveed [45]. A search pattern allows to identify whether any two queries are generated from the same keyword or not. In [39], the authors present a scheme which hides the search patterns that is based on a grouping-based construction to transform the underlying SSE scheme into a new one.

As for the cryptographic systems discussed, both CryptDB and Cipherbase rely on homomorphic encryption, which incurs in an overhead that is only acceptable when the data set is medium-sized [49], and other deterministic cryptographic schemes that leak information in the same fashion as discussed above. Due to using specific trusted hardware that might improve the performance overhead, Cipherbase also suffers from portability issues. Cuttlefish doesn't suffer from this issue, despite also relying on trusted hardware (Intel's SGX), because it alternatively allows computations to be performed over encryption schemes such as the ones used by CryptDB and Cipherbase. Nonetheless, by allowing computations to be performed through encryption schemes or trusted hardware, Cuttlefish suffers from the vulnerabilities of both solutions. Lastly, DepSky is purely a storage solution that doesn't support SSE, thus reducing its practicality.

2.3 Data Stores and Models

In this section we will focus on Not Only SQL (NoSQL) stores, discussing their data models and briefly highlighting some of the most prominent solutions under each category by summarizing their characteristics focusing on their security properties.

We particularly focus on Conflict-free Replicated Data Types (CRDT's) as they can be used as the underlying representation of the data stored in NoSQL stores, a possibility

we intend to explore in this thesis.

2.3.1 NoSQL Models and Stores

Analogous as to what is the definition of NoSQL, the classification concerning the NoSQL data models is not agreed upon. We will adopt the classification and terminology defined by Hetch et. al. [33]. Accordingly there are four main categories — key-value, column-family, document and graph.

2.3.1.1 Key-Value Stores

Key-value stores resemble map or dictionaries. The data is stored in key-value pairs $< k, v >$, where k uniquely identifies v thus allowing to retrieve it or store it. Key-value stores provide a schema-free data model as the value is completely opaque to the data store which means it can hold arbitrary data. This characteristic has the following consequences:

- Key-value stores are not suitable when relations and/or structure is required of the data. These requirements must be met with the necessary logic at the client side application;
- They do not support data-level querying nor indexing. Queries are enabled only through the keys.

Additionally, key-value stores can also be classified as in-memory, which keep the data mainly in memory, or as persistent, which keep the data on disk [32]. In spite of keeping the data mainly in memory, this doesn't mean that in-memory key value stores aren't not persistent as data can be periodically backed up into the disk.

Redis [53] and Riak [54] are examples of such stores.

Redis Redis is an in memory key-value store that allows backups of the data to be made to the disk. It uses asynchronous replication of data, although it is possible to configure synchronous replication, and provides eventual consistency. It offers some data structures such as lists, hashes and sets to store data [1]. Redis is designed to be accessed by trusted clients inside trusted environments and as such doesn't provide many security capabilities. Nonetheless, a weak form of authentication can be configured by having Redis only accept connections from clients who provide a correct password which is set by the administrator in plain text in the Redis configuration files. The security of the connection is left to the client and the selection of a suitable password is of the responsibility of the administrator.

[1] Some of these data structures allow for more refined queries due to secondary indexes, which makes the classification of Redis rather ambiguous as, although claimed to be a key-value store, one can also classify it as a document store.

Riak Riak is a persistent key-value store which uses as asynchronous replication and it can provide eventual or strong consistency. It offers data structures with built-in conflict detection and resolution such as counters, sets and maps which essentially are CRDTs. Unlike Redis, it provides some authorization and authentication mechanisms, like access control and password based authentication, as well as secure communication channels in client-to-server communication.

2.3.1.2 Document Stores

In document stores, such as MongoDB [70], the data is stored in key-document pairs $< k, doc >$, where k uniquely identifies documents within the store. In this sense, document stores are similar to key-value stores. The documents support more complex data including secondary indexes within a document, different schemas of documents in the store, nested documents or lists. As such document stores support data-level query and indexing but, like key-value stores, they do not support relations among documents which makes them unsuitable for these scenarios.

MongoDB MongoDB is a document store which uses as asynchronous replication and it can provide eventual or strong consistency. Unlike the solutions discussed so far, MongoDB offers more extensive security mechanisms [44] thus being quite comparable to relational databases in this regard. It offers mechanisms for authentication, authorization, client-to-server and server-to-server secure communication channels for free while the mechanisms for auditing and encryption of the stored data are only available on its enterprise edition.

2.3.1.3 Column-family Stores

In these types of stores, data is stored in a column-oriented way, therefore one can think of them as tables. Column-stores, e.g. Cassandra [5], follow the guidelines of the data model used in Google's BigTable [17], where the table is analogous to a sparse, distributed, persistent multidimensional sorted map.

A data item is represented by a row, which possesses one or more different columns that in turn can belong to different column families. Each row and column family are uniquely identified by a key, thus allowing more complex queries and indexing. Nonetheless, if the relationships among data are required, they must still be provided by the client application.

Column-stores resemble relational databases when it comes to their representation, but they differ when null values must be handled. Where relational databases would store a null value for each column where no value was provided for, a column family store will only insert a null value in a row for a column if it is required to do so. In other words, homogeneity within a dataset is not enforced thus data items may not have certain attributes specified at all.

Cassandra Cassandra is a column-family store, where instead of rows there is partitions, that uses asynchronous replication and which can provide eventual or strong consistency. It offers its own query language, the Cassandra Query Language (CQL), which supports a rich set data types. It offers security mechanisms for authentication, authorization, client-to-server and server-to-server secure communication channels.

2.3.1.4 Graph Stores

In contrast to the previously discussed stores, graph stores such as Neo4j [46] are specialized on efficient management of highly correlated data. They rely on graph theory and as their data model they use a graph which is a set of nodes and the links that connect them. In these stores the nodes and links have properties, respectively, which are represented by key-value pairs.

Due to be specialized for correlated data, these stores do not scale very well because of the overhead imposed by traversing the relations across nodes in different servers.

Neo4j Neo4j is graph store which uses asynchronous replication and it can provide eventual or strong consistency. It offers its own query language, the Cypher Query Language (Cypher), which supports queries through pattern matching of nodes and relationships in the graph. It offers security mechanisms for authentication, authorization, client-to-server and server-to-server secure communication channels. Additionally, it offers facilities to provide mechanisms for auditing through logs.

2.3.1.5 Critical Analysis

Although they were designed to manage massive amounts of data, the characteristics of these stores must be taken into consideration when choosing which one to use.

Key-value, document and column family gained advantages in distribution by denormalizing the stored data. However, if relationships among data is required, they must be implemented at the client side application which increases its code complexity and it may provoke performance penalties due to the amount of queries that may be necessary. In these situations, graph stores are more suitable. Another important aspect is the amount of expressiveness allowed in queries, even when relationship among data is not required. Typically, key-value stores only have basic operations (i.e. get, put, remove). Document and column-family stores extend these basic operations by allowing range queries, "in"and "and/or". Graph stores are the most expressive as they enable queries to be made using different query languages which leverage different techniques such as pattern matching and graph transversal [33].

Overall, NoSQL data stores have the following issues [32]:

- Low level query languages. As previously discussed, NoSQL don't offer rich queries capabilities;

- Lack of standardization of APIs. The offered APIs are specific to each store, which makes scenarios where more that one NoSQL store could be combined to offer the best solution difficult to achieve;

- Limited security measures. When compared to relational databases, NoSQL stores offer limited security measures. Relational databases typically offer mechanisms for authentication, authorization, auditing and several levels of encryption such as encryption of the data that is stored on the disks (data at rest), client-to-server communication and server-to-server connection encryption. NoSQL usually only offer a subset of these mechanisms.

2.3.2 Conflict-free Replicated Data Types

A Conflict-free Replicated Data Type (CRDT) [64] is a data abstraction which provides guarantees of safety, liveness and convergence to a correct common state upon replication in a self-stablising manner without need for synchronization, despite any number of failures. To ensure absence of conflict, simple mathematical properties such as monotonicity in a semi-lattice and/or commutativity are leveraged.

Replication of CRDT objects is done under Strong Eventual Consistency (SEC). An object is strongly eventually consistent if it is eventually consistent, i.e. all replicas reach the same final value if clients stop submitting updates, and if correct replicas that have delivered the same updates have equivalent state.

The system where the replication of CRDTs occurs, is assumed to have the following properties:

- There is a finite set $\Pi = p^0, ..., p^{n-1}$ of non-byzantine processes;

- Processes in Π may crash silently; a crashed process may remain crashed forever, or may recover with its memory intact. A non-crashed process is said correct;

- The processes of the system are interconnected by an asynchronous network;

- The network can partition and recover.

Similarly to other objects, CRDTs allow for state-based or operation-based replication. As such, they may be specified with respect to both.

State-based object An state based object is a tuple (S, s^0, q, u, m). The replica at process p_i has state $s_i \in S$, called its payload; the initial state is s^0. A client of the object may read the state of the object via query method q and modify it via update method u. Method m serves to merge the state from a remote replica. Any method executes at a single replica. A method whose precondition is satisfied is said enabled and executes as soon as it is invoked.

Every replica occasionally sends its local state to some other replica, which merges the state thus received into its own state. In this way, every update eventually reaches every replica, either directly or indirectly.

Under the assumption of eventual delivery and termination, a state-based object is said to be strongly eventually consistent if it also is a monotonic semilattice object. A monotonic semilattice object is a state-based object equipped with a partial order $\leq$, thus noted as $(S, \leq, s^0, q, u, m)$, with the following properties:

1. The set S of payload values forms a join semilattice ordered by $\leq$. A join semilattice is a partially ordered set that has a least upper bound for any nonempty finite subset. In other words, it has an element s which is the least element that is greater than or equal to all elements of S;

2. Merging sate s with remote state s' computes the least upper bound of the two state;

3. The state is monotonically non-decreasing across updates.

Operation-based object An op-based object is a tuple (S, s^0, q, t, u, P), where S, s^0 and q have the same meaning as in a state-based object. An op-based object has no merge method. As such, an update is split into a pair (t, u), where t is a side-effect-free prepare-update method and u is an effect-update method.

The prepare-update executes at the single replica where the operation is invoked (its source). At the source, prepare-update method t is followed immediately by effect-update method u. The effect-update method executes at all replicas (said downstream). The source replica delivers the effect-update to downstream replicas using a communication protocol specified by the delivery relation P.

Under the assumption of causal delivery of updates and method termination, an op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by causal delivery is said to be strongly eventually consistent.The commutativity property is defined as follows. Updates (t, u) and (t', u') commute if and only if for any reachable state s where both u and u' are enabled:

1. u and u' remain enabled in states $s \bullet u'$ and $s \bullet u$, respectively;

2. $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$ [2].

2.4 Summary

With our research we conclude that although there are solutions that follow a somewhat similar approach to the one we intend to pursuit, like CryptDB [51] and Cipherbase [6] by storing and computing on encrypted data and Cuttlefish [59] by offering secure data

[2] In this context two states s and s' are said to be equivalent, $s \equiv s'$, if all queries return the same result for s and s'. Note that a query has no side-effects which means $(s \bullet q) \equiv s$.

types, there is still room for improvement and to explore new paths to build secure cloud and distributed computing solutions.

Computing over encrypted data, using mechanisms like homomorphic and searchable encryption schemes, allows to mitigate data privacy issues in the cloud and other distributed computing environments. For its practical use, factors like the operation's performance requirements and the implications of the data information leakage for the cloud base application must be taken into consideration. Fully Homomorphic encryption remains unpractical due to its performance overhead yet partial and somewhat homomorphic achieve better performance at the cost of limiting the type and the number of operations allowed. Searchable symmetric encryption reduces the performance overhead by revealing information about the queried data.

NoSQL data stores are highly available and perform well in distributed systems such as the cloud. In an approach similar to the Riak [54] store and its data structures, our goal is to design secure CRDTs (SCRDTs) libraries that may be incorporated into a NoSQL data store in order to provide security guarantees over it. The CRDTs can easily be stored under any of the discussed data models and in the case of key-value stores, they may increase query expressiveness if the operations they support are exposed by the data store API.

Chapter 3

Solution Design

As previously mentioned, with this thesis we aim at developing a set of secure abstractions, specifically secure CRDTs or SCRDTs, to support secure computations in the context of NoSQL data stores hosted by cloud services and to serve as a tool for programmers to develop their own distributed applications. In this context, secure means that the data stored in these abstractions are confidentiality and privacy protected. The SCRDTs we designed were implemented with the support of different cryptographic algorithms to provide the intend security.

In this chapter, Section 3.1 provides an overview of the architecture of the system where the SCRDTs are intended to be used, which is a cloud-backed NoSQL distributed data store, and Section 3.2 defines our threat model. Finally, Section 3.3 describes and discusses all of the CRDTs and correspondent SCRDTs developed within this thesis.

3.1 System Architecture

Figure 3.1 illustrates the architecture of an NoSQL distributed data store where the SCRDTs would be integrated in. This store uses the key-value data model where the keys map to the SCRDTs we developed, it is in memory and exports get, put and remove operations. Each node of the system represents different instances, i.e. replicas, of the store service running in the cloud which may or may not be located within the same data center.

Each node is composed by the following components: *Storage*, *Client Interface* and *Replication Mechanism*.

The *Storage* component is where data is effectively stored in one of our SCRDT data structures, which in turn are stored under the key value data model. In other words, the storage component maps to the memory of the machine where the service replica is

running.

The *Client Interface* exports the API necessary to manipulate the data in the storage component. Although the data store model is of the type key value which typically only supports an limited amount of operations (e.g.: put, get and remove operations), through the use of our *Stubs* component which provides direct support of all of the operations of each CRDT and SCRDT it is possible to achieve a more expressive API.

The *Replication Mechanism* is responsible for replicating operations and merging the state of the stored SCRDTs with the state of other replicas in order to ensure their convergence.

The manipulation of the data on the client side is mediated by the *Stubs* component. For each of the structures we designed there is an corresponding Stub that is made available on the client side. This stub is responsible for:

- Exporting the operations supported by its corresponding CRDT and SCRDT abstractions to the client;
- Applying the operations of its corresponding CRDT and SCRDT abstractions and propagating their result to the data store service replica through the use of the get, put and remove operations exported by the data store;
- Encrypting and decrypting data under the appropriate encryption schemes in the case of manipulation of SCRDTs.

Figure 3.2 shows a sequence diagram of an use case of how the *Stub* component intervenes when a client of the system invokes the add operation of an element e over an set identified by the key S in the data store. The encryption scheme to be used is chosen based on the following factors:

1. the invoked operation;
2. the selected structure.

It follows that this component is also responsible for managing the required cryptographic keys that must be supplied by the client. If data is shared among clients, it is assumed that a key sharing protocol was made *a priori*.

Communication among replicas of the service and between clients and replicas of the service, signalled by (1) and (2) in Figure 3.1 respectively, are assumed to be made over secure communications channels such as a TLS channel.

3.2 Threat Model

As we aim the data store system is suported by cloud services, our threat model is comprised by a cloud administrator adversary of the type Honest but Curious (HbC) [19].

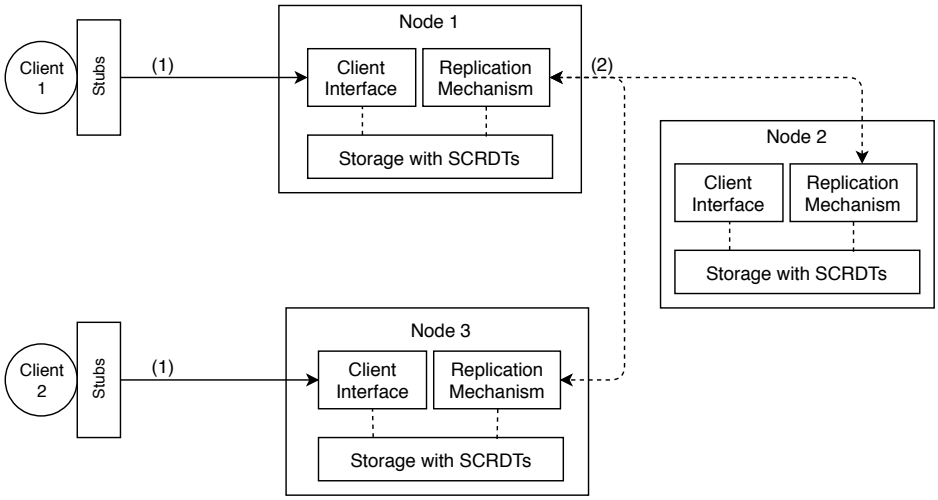

Figure 3.1: System model diagram.

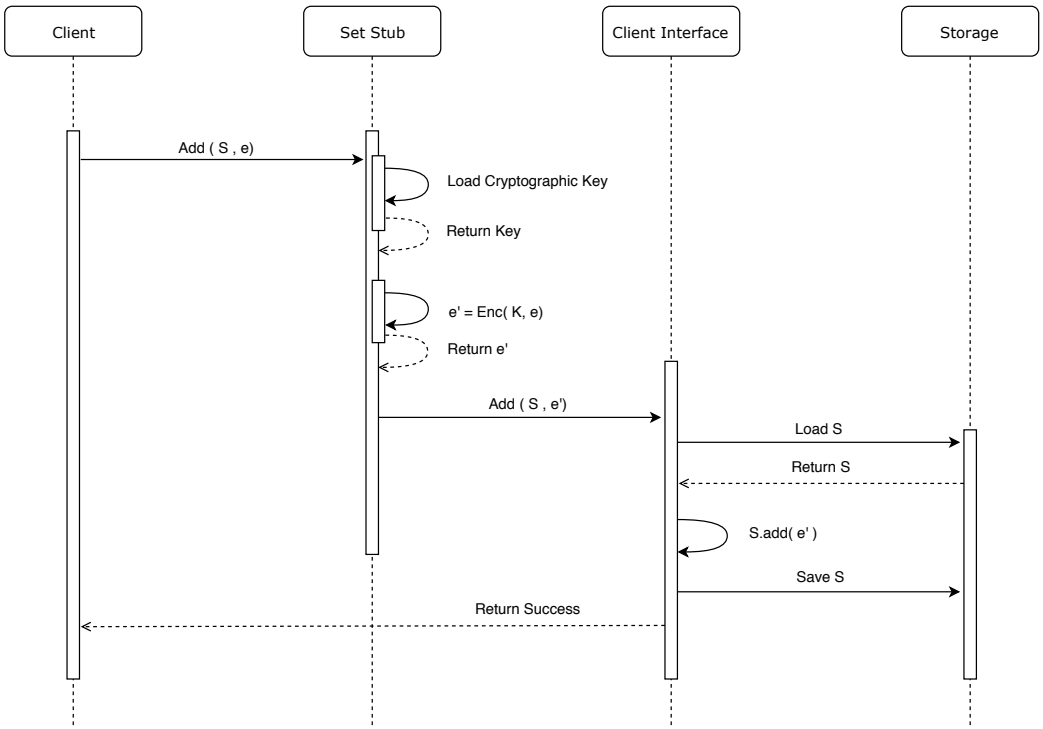

Figure 3.2: Use case of an Stub component In this example, a client wishes to add an element e to an set identified by key S in the data store.

This administrator has access to the cloud provider's infrastructure and software platforms which consequently provides him access to customer data. The administrator main goal isn't to corrupt the computations but to gather as much information about them as possible in order to infer their nature.

We do not consider external attacks to the communication channels over the internet (signalled by (1) and (2) in Figure 3.1) in our threat model as we assume that all communications are made through the use of secure communication channels such as a TLS channel. External in this context refers to the fact that the attacks are performed by adversaries external to the cloud provider's infrastructure.

3.3 Data Structures

To develop our solution, firstly we studied and then selected which of the existent CRDTs in the literature [52, 63] were the most relevant and important to provide support to data stores and web applications. These where registers, sets, lists, maps and counters. Regarding the replication method and concurrency conflict resolution polices of the structures, we opted for state based replication to follow a similar approach as current industry solutions like Riak [54], and the following policies as they are the most common ones and are valuable in different use case scenarios:

1. Add Wins: given two concurrent update operations of adding and removing an element to the structure, the add operation persists.

2. Last Writer Wins: given an total order relationship over the update operations, the update that persists is the one issued by the last writer.

Table 3.1 maps each of the selected data structures in each of the policies.

As our goal was to build secure versions of CRDTs, the SCRDTs, by supporting all of their operations with the aid of cryptographic algorithms, we analysed which of these algorithms enabled to support said operations with the maximum security possible for each of the selected CRDTs. Table 3.2 resumes the results of this analysis.

In the following sections a detailed description and specification of each of the CRDT data structures is provided along with the methodology used in selecting which cryptographic algorithms will support their respective SCRDT version. The main difference between SCRDTs and CRDTs is that the former stores encrypted data while the latter stores non encrypted data, thus no specification is provided for the SCRDTs as it would be the same as its correspondent CRDT. Finally, Section 3.3.6 provides a critical analysis regarding the security of the SCRDTs developed.

3.3.1 Set

The set structure behaves just like an mathematical set, i.e. it is a collection of distinct elements. For this structure, we developed three variants — grow only, last writer wins

Table 3.1: Mapping of CRDTs and conflict resolution policies.

CRDT	Conflict Resolution Policies
Set	Last Writter Wins, Add Wins
List	Last Writter Wins, Add Wins
Map	Last Writter Wins, Add Wins
Register	Last Writter Wins
Counter	Non Aplicable

Table 3.2: Mapping of CRDTs and cryptographic algorithms.

CRDT	Cryptographic Algorithm
Set	Deterministic Encryption
List	Deterministic Encryption
Map	Deterministic and Probabilistic Encryption
Register	Probabilistic Encryption
Counter	Partial Homomorphic Encryption

and add wins set —that have the following operations:

1. Add: inserts an element in the set;
2. Remove: removes an element from the set;
3. Contains: checks if a given element belongs to the set;
4. Size: returns the number of elements of the set;
5. Getall: returns all the elements within the set.

Operations 1 and 2 are categorized as updates as they modify the structure and the rest of the operations are categorized as queries. Next we detail three variants of the set CRDT that we developed.

Grow Only Set (G-Set) The grow only set (Algorithm 1), is an exception within the developed sets as it only one update operation, the add operation. This property alongside the uniqueness property of the elements within a set results in the impossibility to have concurrency conflicts regarding the add operation. Despite being quite simple, this set is an important pillar stone to build other CRDTs that are more complex. The state of an G-Set is represented by a set S of elements e. The contains operation simply tests if an element e belongs to the state S and the merge of two G-Sets consists in the union of their respective states.

Last Writer Wins Set (LWW-Set) The last writer wins set (Algorithm 2) is similar to the specification given by Shapiro et. al. [63]. As in this set there are two update operations that might lead to a concurrency conflict, there is a need for an conflict resolution policy. In this case, the policy is the Last Writer Wins. To enforce this policy additional information, namely a time stamp ts, is kept alongside the elements within the state of the LWW-Set.

To support the remove operation, the state of the set is represented by two distinct G-Sets, sets A and R. Whenever an element e is added, or removed, to the LWW-Set at an instant ts, the pair (e, ts) is inserted in A, or R. An element e is said to be contained in set if:

1. $(e, \_) \in A \land (e, \_) \notin R$, or
2. $(e, ts) \in A \land (e, ts') \in R \land ts > ts'$.

As the state of an LWW-Set is represented by two G-Sets, the merge operation between two LWW-Sets consists in merging the G-Sets that represent their state.

Add Wins Set The add wins set (Algorithms 3 and 4), also known as observed-removed set [63], also allows the remove operation. However in this set this operation is supported through the use of time stamps ts and unique identifiers id which are associated to each of its elements within its state and, as indicated by the name, the concurrency conflict resolution policy is Add Wins. Additionally and to enforce the Add Wins policy, a flag $type$ is also kept to indicate which was the last operation made over the element.

The state representation of an Add Wins Set is a map of associations of type $id \rightarrow (e, ts, type)$. Whenever an element e is added to this set at an instant ts, an unique identifier id is generated and an association $id \rightarrow (e, ts, type)$ is inserted in the state. As a consequence, we have that an element may be added multiple times to the set.

Whenever an element e is removed at an instant ts, all of the associations $id \rightarrow (e, ts', type)$ in the state for which the elements match and the time stamp of the remove operation is bigger, i.e when $e = e' \land ts > ts'$, are marked as removed. In other words, all of the add operations up until the instant of the remove operation are eliminated.

An element belongs to the set if there is an identifier in the state that maps to a tuple containing this element and if the said tuple hasn't been marked as removed. Therefore, although an element may be added to the set under different identifiers, the element is only visible within the set once.

Unlike the other operations of this set, the contains operation is supported through an additional data structure. This data structure, represented by the variable *mapper* in Algorithms 3 and 4, maps each of the elements of the Add Wins set in the set of all the elements identifiers. This design choice, which trades storage space and complexity for performance, avoids iterating over the whole state to find an entry that complies to the condition of an element belonging to the set.

Algorithm 1 Grow Only Set

```
1:  function GSet( )
2:      state ← {}
3:
4:  function Add( elem )
5:      state ← state ∪{elem}
6:
7:  function GetAll( )
8:      return state
9:
10: function Contains( elem )
11:     if elem ∈ state then
12:         return true
13:     else
14:         return false
15:
16: function Size( )
17:     return #state
18:
19: function Merge( otherstate )
20:     state ← state ∪ otherstate
```

3.3.1.1 Set SCRDT and Stub

For all the described set variants there is an implicit requirement to perform equality comparisons based on the value of an element to comply with the uniqueness property of elements within a set. Specifically, it is a frequent necessity to verify if an element belongs the state of the set variants.

Accordingly, to create a secure version of each of the variants it must be possible to execute this comparison in the ciphertext. For this purpose, we leveraged property preserving encryption algorithms to build the said version. In particular, we used deterministic encryption to cipher each of the elements that belong to the sets. Algorithm 5 shows the pseudocode for the stub of all of the set's SCRDTs.

Algorithm 2 Last Writer Wins Set

```
1:  function LWWSet( )
2:      addSet ← GSet()
3:      removeSet ← GSet()
4:
5:  function Add( elem, ts )
6:      addSet[elem] ← ts
7:
8:  function Remove( elem, ts)
9:      removeSet[elem] ← ts
10:
11: function Cointains( elem )
12:     if elem ∈ Filtered() then
13:         return true
14:     else
15:         return false
16:
17: function GetAll( )
18:     return Filtered()
19:
20: function Size( )
21:     return #Filtered()
22:
23: function Merge( otherAddSet, otherRemoveSet )
24:     addSet.merge( otherAddSet )
25:     removeSet.merge( otherRemoveSet )
26:
27: function Filtered( )                          ▷ Auxiliar method
28:     r ← {}
29:     for (elem, ts) ∈ addSet do
30:         if (elem, _) ∉ removeSet then
31:             r ← r ∪{elem}
32:         else
33:             (elem', ts') ← (e,t) ∈ removeSet ∧ elem' = elem
34:             if ts > ts' then
35:                 r ← r ∪{elem'}
```

Algorithm 3 Add Wins Set - Part I

```
1:  function AddWinsSet( )
2:      elemns ← {}                                   ▷ The actual state representation
3:      mapper ← {}                  ▷ Map of elements and the id's associated with them
4:
5:  function Add( id, elem, ts )
6:      if elemns[id] =⊥ then                          ▷ An id we have yet to observe
7:          elemns[id] ← (elem, ts, 1)
8:          if mapper[elem] ≠⊥ then
9:              mapper[elem] ← mapper[elem] ∪{id}
10:         else
11:             mapper[elem] ←{id}
12:     else                                           ▷ An id we have observed
13:         (elem', ts', _ ) ← elemns[id]
14:         if elem' = elem ∧ ts' ≤ ts then   ▷ Checking if it's associated with this element
15:             elemns[id] ← (elem, ts, 1)
16:
17: function Remove( elem, ts)
18:     idSet ← mapper[elem]
19:     if idSet ≠⊥ then
20:         for all id ∈ idSet do
21:             (_, ts', _ ) ← elemns[id]
22:             if ts' < ts then
23:                 elemns[id] ← (elem, ts, 0)
24:
25: function Cointains( elem )
26:     idSet ← mapper[elem]
27:     if idSet ≠⊥ then
28:         for all id ∈ idSet do
29:             ( _ ,_ , type ) ← elemns[id]
30:             if type = 1 then
31:                 return true
32:     return false
33:
34: function GetAll( )
35:     r ← {}
36:     for all (_, elem, _, type) ∈ elemns do
37:         if type ≠ 0 ∧ elem ∉ r then
38:             r ← r ∪{elem}
39:     return r
40:
41: function Size( )
42:     return #GetAll()
```

Algorithm 4 Add Wins Set - Part II

```
1: function Merge( otherState )
2:     for (id, elem, ts, type) ∈ otherState do
3:         if type = 1 then
4:             Add(id,elem, ts)
5:         else if type = 0 then
6:             Remove (id,elem, ts)
```

Algorithm 5 Set Stub

```
1: function SetStub( store, pwd, iv )
2:     K_d ← loadKey(store, pwd);
3:     IV ← iv;
4:     service ← getServiceEndpoint();
5:
6: function Add( Key, elem )
7:     elem' ← detEnc(K_d, IV , elem)
8:     service.setAdd(Key, elem')
9:
10: function Remove( Key, elem )        ▷ This operation is non applicable for the G-Set
11:     elem' ← detEnc(K_d, IV , elem)
12:     service.setRemove(Key, elem')
13:
14: function Contains( Key, elem )
15:     elem' ← detEnc(K_d, IV , elem)
16:     return service.setContains(Key, elem')
17:
18: function GetAll( Key )
19:     encState ← service.setGetAll(Key)
20:     state ←{}
21:     for e ∈ encState do
22:         e' ← detDec(K_d, IV , e)
23:         state ← state ∪{e'}
24:     return state
25:
26: function Size( Key )
27:     return service.setSize(Key)
```

3.3.2 List

The list structure behaves like an mathematical sequence, i.e it is an collection of elements that allows for repetitions and which is enumerated. The list CRDT we developed is built upon a modified version of the CRDT TreeDoc proposed by Preguiça et. al. [52] and has the following operations:

1. Insert: inserts an element in the specified position of the list;
2. Remove: removes an element in the specified position from the list;
3. Contains: checks if a given element belongs to the list;
4. Get: returns the element stored in the list in the specified position;
5. Size: returns the number of elements of the list;
6. Getall: returns all the elements within the list.

Operations 1 and 2 are categorized as updates as they modify the structure and the rest of the operations are categorized as queries. Most of these operations are supported through the use of indexes which in a distributed environment, where there is different copies of the list that must present an unique and consistent order of its elements, might lead to a problem if distinct elements are concurrently inserted in the same position in distinct copies of the list. In such scenario, upon merging these distinct copies it is impossible to deterministically order these insertions using only the index in order to compute the final and correct state for both lists. The TreeDoc CRDT addresses this concern.

Originally, TreeDoc consists in a sequential shared buffer where each entry of the vector is associated with an identifier $PosId$ with the following properties:

1. Every element in the buffer has identifier;
2. There is a total order of the identifiers, represented by $<$ that is consistent with the order of the elements within the buffer.
3. Given any identifiers id_1 e id_2, it is always possible to define a new identifier id_3 between id_1 e id_2.

These properties are required to ensure that concurrent updates commute thus ensuring convergence of replicas. The identifier $PosId$ consists in the pair (p,d) where p is the position the element was intended to be inserted in and d is an disambiguator. The purpose of this disambiguator is to enable to establish a total order among elements that were inserted in the same position. It follows that the state representation of the TreeDoc is an ordered list of $(PosId,e)$.

In our list (Algorithms 6 and 7) take leverage the identifiers, where the disambiguator is of the type simple disambiguator (SDIS) [52], and we associate them to the element plus some metadata, namely a time stamp ts and the type $type$ of the update operation that was last performed, similarly to our approach in the Add Wins Set, to help enforce conflict resolution policies. Thus the state of our list is represented by a map of entries of the type $PosId \rightarrow (e, ts, type)$.

We also maintain two additional data structures to help support the list's operations — $atoms$ and ids. The ids structure is an array that in each entry i contains the identifier $PosId$ of the element that is currently at that position in the lists and it only contains the identifiers of the elements that effectively are in the list. The $atoms$ structure is a map of $PosId \rightarrow e$ entries that maps all of the identifiers of the elements that effectively are in the list in their elements. These structures allow for better performance of the list's query operations (getAll, get, contains) and the remove operation at a cost of using more memory.

We implemented two variants of the list: add wins and last writer wins. As their behaviour is similar except for the remove operation, the main algorithm for the list is shown in Algorithms 6 and 7 whereas the remove operations of the last writer wins and add wins variants is shown in Algorithms 8 and 9, respectively.

Last Writer Wins List (LWW-List) The Last Writer Wins list, as the name indicates, applies the last writer wins concurrency conflict resolution policy. Whenever an element e is inserted in the list at position i at instant ts, an identifier $PosId = (i, d)$ is generated where the disambiguator d will take a value that is between the disambiguators of the identifiers in positions $i - 1$ and $i + 1$ of the list. After creating the identifier, an entry $PosId \rightarrow (e, ts, type)$ is inserted in the state, a pair $(PosId, e)$ is inserted in $atoms$ and $PosId$ is inserted in ids.

Whenever a remove is issued for position i in the instant ts, the element to be removed is selected based on the identifier that is in the position i in ids structure. Afterwards, if the time stamp ts' of the last operation associated with this element is bigger or equal than ts, the element is marked as removed and the auxiliary data structures $atoms$ and ids are invalidated so that they may be recomputed again. An element is said to be in the list if it is present in the $atoms$ structure. The merge operation consists in the union of the state of the two lists, where for the elements stored in the same position the process described for the insert operation is used.

Add Wins List The Add Wins list, again as the name indicates, applies the add wins concurrency conflict resolution policy and behaves like the LWW-List in every operation except the remove. In this list, whenever a remove is issued for position i in the instant ts, the element to be removed is selected based on the identifier that is at position i in ids. Afterwards if the time stamp ts' of the last operation associated with this element is strictly bigger than ts, the element is marked as removed and the auxiliary data structures

atoms and *ids* are also invalidated. If the time stamps *ts* and *ts'* are equal then the element is only marked as removed if the type *type* of the last operation associated with the element is not an insertion.

3.3.2.1 List SCRDT and Stub

Despite the majority of the operations of the list being supported through indexes, the contains operation is based upon the value of the element. Consequently, for the list there is a requirement to be able to perform equality comparisons, as with the set. Specifically, in this structure it is necessary to verify if the element belongs to the *atoms* auxiliary structure.

Accordingly, the secure versions of the list's variants store their elements under a deterministic encryption scheme. Algorithm 10 shows the pseudocode for the stub of all of the list's SCRDTs.

Algorithm 6 List - Part I

```
1:  function List( )
2:      state ← {}
3:      atoms ← {}                          ▷ Ordered map of position id's and elements
4:      ids ← {}                                                  ▷ Ordered set of id's
5:
6:  function Insert( elem, ts )
7:      Insert( Size(), elem, ts )
8:
9:  function Insert( i, elem, ts )
10:     id ← CreatePosition( i )                          ▷ Creates an composite identifier
11:     if state[id] ≠⊥ then
12:         (elem, ts', type) ← state[id]
13:         if ts ≥ ts' then
14:             state[id] ← (elem, ts, 1)                      ▷ Inserts are operation type 1
15:             atoms ← atoms ∪{ (id, elem) }
16:             ids ← ids ∪{ id }
17:     else
18:         state[id] ← (elem, ts, 1)
19:         atoms ← atoms ∪{ (id, elem) }
20:         ids ← ids ∪{ id }
21:
22: function Remove( ts )
23:     return Remove( Size()-1, ts )                          ▷ See algorithms 8 and 9
24:
25: function Get( i )
26:     idArray ← GetIdsArray()                ▷ Computes and returns all of the ids
27:     atoms ← GetAtoms()                     ▷ Computes and returns all of the atoms
28:     id ← idArray[i]
29:     elem ← atoms[id]
30:     return elem
31:
32: function Cointains( elem )
33:     atomsMap ← GetAtoms()
34:     for all (id, elem') ∈ atomsMap do
35:         if elem = elem' then
36:             return true
37:     return false
38:
39: function GetAll( )
40:     atomsMap ← GetAtoms()
41:     r ← {}
42:     for all (id, elem) ∈ atomsMap do
43:         r ← r ∪{elem}
44:     return r
45:
46: function Size( )
47:     return #GetAtoms()
```

Algorithm 7 List - Part II

```
1: function MERGE( otherState )
2:     for all (id, elem, ts', type) ∈ otherState do
3:         i ← id.GetPosition()
4:         if type = 1 then
5:             Insert(i, elem, ts)
6:         else if type = 0 then
7:             Remove(i, elem, ts)
```

Algorithm 8 Last Writer Wins

```
1: function REMOVE( i, ts )
2:     idArray ← GetIdsArray()
3:     id ← idArray[i]
4:     if state[id] ≠ ⊥ then
5:         (elem, ts', type) ← state[id]
6:         if ts ≥ ts' then
7:             state[id] ← (⊥, ts, 0)                    ▷ Removes are operation type 0
8:             atomsArray ← getAtoms()
9:             elem ← atomsArray[id]
10:            Delinearize()                  ▷ Invalidates the atoms and id's structures
11:            return elem
12:    return ⊥
```

Algorithm 9 Add Wins List

```
1: function REMOVE( i, ts )
2:     id ← GetIdsArray().get( i )
3:     if state[id] ≠ ⊥ then
4:         (elem, ts', type) ← state[id]
5:         if ts > ts' ∨ (ts' = ts ∧ type ≠ 1) then
6:             state[id] ← (⊥, ts, 0)                    ▷ Removes are operation type 0
7:             atomsArray ← getAtoms()
8:             elem ← atomsArray[id]
9:             Delinearize()                  ▷ Invalidates the atoms and id's structures
10:            return elem
11:    else
12:        state[id] ← (⊥, ts, 0)
```

Algorithm 10 List Stub

```
1:  function ListStub( store, pwd, iv )
2:      K_d ← loadKey(store, pwd);
3:      IV ← iv;
4:      service ← getServiceEndpoint();
5:
6:  function Add( Key, elem)
7:      elem' ← detEnc(K_d, IV , elem)
8:      service.listInsert(Key, elem')
9:
10: function Add( Key, i , elem)
11:     elem' ← detEnc(K_d, IV , elem)
12:     service.listInsert(Key, i, elem')
13:
14: function Remove( Key )
15:     service.listRemove(Key )
16:
17: function Remove( Key, i )
18:     service.listRemove(Key, i)
19:
20: function Contains( Key, elem )
21:     elem' ← detEnc(K_d, IV , elem)
22:     return service.listContains(Key, elem')
23:
24: function Get( Key, i )
25:     elem ← service.listGet(key, i)
26:     if elem ≠⊥ then
27:         return detDec(K_d, IV , elem)
28:     return ⊥
29:
30: function GetAll( Key )
31:     encState ← service.listGetAll(key)
32:     state ← {}
33:     for e ∈ encState do
34:         e' ← detDec(K_d, IV , e)
35:         state ← state ∪{e'}
36:     return state
37:
38: function Size( Key )
39:     return service.listSize(Key)
```

3.3.3 Register

A register behaves like a container for some arbitrary value which can be updated at any given time. The register CRDT implemented (Algorithm 11) is similar to the specification given by Shapiro et. al. [63], which enforces the last writer wins conflict policy and has the following operations:

1. Put: updates the value stored in the register;
2. Get: returns the value stored in the register;

Operation 1 is categorized as an update as it modifies the structure's state and operation 2 is categorized as a query.

The register CRDT stores a time stamp ts, besides arbitrary value v, that indicates when the last update to the value was made thus its state is represented by the pair (v, ts). A put operation of value v' issued on the instant ts' consists in updating the value stored within the register if the time stamp ts is greater or equal the current time stamp stored in the register, i.e it verifies if $ts \geq ts'$. The get operation is very straightforward and simply returns the current value of the register. The merge operation between two registers determines which of the values should persist by applying the same logic of the put operation.

Algorithm 11 Last Writer Wins Register

```
1:  function LWWRegister( )
2:      state ← (⊥, ⊥)
3:
4:  function Put( val, ts )
5:      ( val', ts') ← state
6:      if ts ≥ ts' then
7:          state ← (val, ts)
8:
9:  function Get( )
10:     (val, ts ) ← state
11:     return val
12:
13: function Merge( otherstate )
14:     (val, ts) ← state
15:     (val', ts') ← otherstate
16:     if ts' ≥ ts then
17:         state ← (val', ts')
```

3.3.3.1 Register SCRDT and Stub

As none of the register operations is made through the value there are no requirements over it, thus the secure version of the register stores the value under a probabilistic

encryption scheme. Algorithm 12 shows the pseudocode for the stub of the register SCRDT.

Algorithm 12 Register Stub

```
1:  function RegisterStub( store, pwd, iv )
2:      K_r ← loadKey(store, pwd);
3:      IV ← iv;
4:      service ← getServiceEndpoint();
5:
6:  function Put( Key, v )
7:      v' ← rndEnc(K_r, IV , v)
8:      service.registerPut(Key, v')
9:
10: function Get( Key )
11:     v ← service.registerGet(Key)
12:     if v ≠⊥ then
13:         return rndDec(K_r, IV , v)
14:     return ⊥
```

3.3.4 Map

A map is a collection of (key, value) pairs where the key must be unique within the collection. The map CRDT has the following operations:

1. Put: inserts a key value entry in the map;
2. Remove: removes the entry mapped by the given key from the map;
3. Contains: checks if there is an entry mapped by the given key;
4. Get: returns the value in the entry that is mapped by the given key;
5. Size: returns the number of elements of the map;
6. Getall: returns all the elements within the map.

Operations 1 and 2 are categorized as updates as they modify the structure and the rest of the operations are categorized as queries. As with the list, for the map two variants were developed: last writer wins and add wins map.

Last Writer Map (LWW-Map) In the last writer wins map (Algorithms 13 and 14), the state consists in a map of entries $k \rightarrow rg$, where rg is a register CRDT which enforces the last writer wins policy and is detailed in Section 3.3.3. This composition of CRDTs enables to simplify the logic required to both enforce the conflict policy and to perform the merge operation.

The put operation of a pair (k, v) at an instant ts consists in inserting the entry $k \rightarrow rg$ in the state. The remove operation of a value v associated with a key k at an instant ts consists in inserting the null value in the register rg associated with the key k so that this entry is marked as removed. A key k is said to be in the map if there is an entry for that key in the state and if the value stored in the register associated with this key is not null. The merge operation between two LWW-Maps is the union of the two maps that represent their state where for the keys common to both maps, the merge operation of the associated registers is invoked.

Add Wins Map In the add wins map (Algorithms 15 and 16) the state consists in a map of entries $k \rightarrow (e, ts, type)$, where ts is the time stamp of the last update performed in this entry and $type$ indicates which was the last update made. For this map, the put operation behaves just as described for the LWW-Map. The remove operation of a value v associated with a key k at an instant ts consists in marking the tuple $(e, ts', type)$ associated with the key in the map's state as removed if the time stamp of the remove operation ts is strictly bigger to the time stamp ts' of the last operation performed over this entry. If the time stamps are equal, then the remove is only effective if the type of the last performed operation isn't a put. A key k is said to be in the map if there is an entry for that key in the state and if its associated tuple is not marked as removed. The merge between two Add Wins maps is the union of the two maps that represent their state, where for the keys common to both maps the last operation performed is maintained if there are no conflicts. If there are conflicts, the same logic used in the put operation is applied to enforce the Add Wins policy.

3.3.4.1 Map SCRDT and Stub

For the maps, as with the sets and lists, there is also an implicit requirement to be able to perform equality comparisons. However, in the maps the requirement is only applicable to the key as it is used to support the majority of its operations. Therefore, the key must be stored under a deterministic encryption scheme while the value may be stored under a probabilistic encryption scheme as there are no requirements for it to oblige to. Algorithm 17 shows the pseudocode for the stub of all of the map's SCRDTs.

Algorithm 13 Last Writer Wins Map - Part I

```
1:  function LWWMap( )
2:      state ← {}
3:
4:  function Put( key, val, ts )
5:      rg ← state[key]
6:      nRg ← LWWRegister(val,ts)
7:      if rg =⊥ then
8:          state[key] ← nRg
9:      else
10:         rg.merge( nRg )
11:
12: function Remove( key, ts )
13:     rg ← state[key]
14:     if rg ≠⊥ then
15:         rg.put( ⊥ , ts )
16:
17: function Get( key )
18:     rg ← state[key]
19:     if rg ≠⊥ then
20:         return rg.get()
21:     else
22:         return ⊥
23:
24: function Contains( key )
25:     rg ← state[key]
26:     if rg ≠⊥ then
27:         return true
28:     else
29:         return false
```

3.3.5 Counter

A counter behaves like a register for some arbitrary numeric value which can be either incremented or decremented by some value. The counter CRDT we developed is inspired by the PN-Counter proposed by Shapiro et. al [63] which has the following operations:

1. Inc: increments the current value of the counter by some specified value;
2. Dec: decrements the current value of the counter by some specified value;
3. Get: returns current value of the counter.

Operations 1 and 2 are categorized as updates they modify the counter's state and operation 3 is categorized as a query.

Originally in the PN-Counter there are two vectors that represent the state, vectors P and N, that have as many entries as the number of copies that exist of the counter. Each

Algorithm 14 Last Writer Wins Map - Part II

```
1:  function Size( )
2:      return #Filtered()
3:
4:  function GetAll( )
5:      return Filtered()
6:
7:  function Filtered( )
8:      r ← {}
9:      for (key, rg) ∈ state do
10:         value ← rg.get()
11:         if value ≠ ⊥ then
12:             r ← r ∪{(key, value)}
13:     return r
14:
15: function Merge( otherstate )
16:     for (key, rg) ∈ otherstate do
17:         myRg ← state[key]
18:         if myRg = ⊥ then
19:             state ← state ∪ {(key, rg)}
20:         else
21:             myRg.merge( rg )
```

copy only manipulates one predefined entry i in both arrays and its only possible to make unitary increments and decrements, which are registered in the vectors $P[i]$ and $N[i]$ accordingly.

Specifically, the inc operation issued at a copy that manipulates the entry i consists in incrementing the entry $P[i]$ and the dec operation consists in incrementing the entry $N[i]$. The absolute value of the counter is given by $\sum P[i] - N[i], i = 0, ..., N$ where N represents the total number of copies of the counter. The merge operation between two PN-Counters consists in calculating the maximum value for each entry of both vectors of each state.

In our version of the counter (Algorithm 18), we allow increments and decrements of arbitrary values and, besides the vectors P and N, two auxiliary vectors are also kept, MP and MN , which also have as many entries as the number of copies that exist of the counter and each entry contains a time stamp that tracks when was the last update made to the corresponding entry at vectors P and N respectively.

While the absolute value of the counter is computed exactly the same way as in the PN-Counter, the inc and dec operation are slightly different in the sense that arbitrary values for increments and decrements are allowed and that the auxiliary vectors MP and MN are also modified. Specifically, the inc operation issued at a copy that manipulates the entry i updates the time stamp stored in the entry $MP[i]$ and the dec operation issued at a copy that manipulates the entry i updates the time stamp stored in the entry $MN[i]$.

Algorithm 15 Add Wins Map - Part I

```
1:  function AddWinsMap( )
2:      state ← {}
3:
4:  function Put( key, val, ts )
5:      if state[key] ≠⊥ then
6:          (val', ts', type) ← state[key]
7:          if ts ≥ ts' then
8:              state[key] ← (val, ts, 1)                ▷ puts are operation type 1
9:      else
10:         state[key] ← (val, ts, 1)
11:
12: function Remove( key, ts )
13:     if state[key] ≠⊥ then
14:         (val, ts', type) ← state[key]
15:         if ts ≥ ts' ∨ (ts' = ts ∧ type ≠ 1) then
16:             state[key] ← (⊥, ts, 0)                  ▷ removes are operation type 0
17:
18: function Get( key )
19:     if state[nKey] ≠⊥ then
20:         (val, ts, type) ← state[key]
21:         return val
22:     else
23:         return ⊥
24:
25: function Contains( key )
26:     if state[key] ≠⊥ then
27:         (val, ts, type) ← state[key]
28:         if val ≠⊥ then
29:             return true
30:         else
31:             return false
32:     else
33:         return false
34:
35: function Size( )
36:     return #GetAll()
37:
38: function GetAll( )
39:     r ← {}
40:     for (key, val, ts, type) ∈ state do
41:         if value ≠⊥ then
42:             r ← r ∪{(key, val)}
43:     return r
```

Algorithm 16 Add Wins Map - Part II

```
1: function MERGE( otherstate )
2:     for (key, val, ts, type) ∈ otherstate do
3:         if type = 1 then
4:             Put( key, val, ts )
5:         else if oType = 0 then
6:             Remove( key, ts )
```

3.3.5.1 Counter SCRDT and Stub

The previously explained modifications where made in order to enable the creation of the secure version of the counter. To support the inc and dec operations, an partial homomorphic encryption scheme such as Paillier's [48] is required so that it may be possible to perform additions and subtractions over the ciphertext. Therefore the data at vectors P and N must be stored under such scheme. However, as in the original PN-Counter the merge operation consists in comparing the entries at these vectors in search of the maximum entry, a problem arises as partial homomorphic encryption schemes do not allow for such comparison to be made over its ciphertext.

Thus as an alternative, we created the additional vectors MP and MN to support the merge operation. Specifically, the merge operation between two of our modified counters consists in calculating the maximum value for each entry of vectors MP and MN and maintaining the values of the corresponding entries at vectors P and N. This solution works correctly because each replica only modifies one entry per auxiliary vector and does so in an monotonically increasing way therefore it follows that given two vectors MP and MP', if at an entry i $MP[i] > MP'[i]$ then $P[i] > P'[i]$ (which is also applicable for vectors MN and MN').

At last, allowing for arbitrary increments and decrements permits to obfuscate to an extent an adversary that might be monitoring the computations in search for patterns to learn information, such as a HbC cloud administrator, by hiding the fact that the state of the counter changes in the same way every time an update operation is performed.

Algorithm 19 shows the pseudocode for the stub of the counter SCRDT.

Algorithm 17 Map Stub

```
1:  function MapStub( store, pwd, ivD, ivR )
2:      K_d ← loadKey(store, pwd);
3:      K_r ← loadKey(store, pwd);
4:      IV_d ← ivD;
5:      IV_r ← ivR;
6:      service ← getServiceEndpoint();
7:
8:  function Put( Key, k, v )
9:      k' ← detEnc(K_d, IV_d , k)
10:     v' ← rndEnc(K_r, IV_r , v)
11:     service.mapPut(Key, k',v')
12:
13: function Remove( Key, k )
14:     k' ← detEnc(K_d, IV_d , k)
15:     v ← service.mapRemove(Key, k')
16:     if v ≠⊥ then
17:         return rndDec(K_r, IV_r , v)
18:     return ⊥
19:
20: function Contains( Key, k )
21:     k' ← detEnc(K_d, IV_d , k)
22:     return service.mapContains(Key, k')
23:
24: function Get( Key, k )
25:     k' ← detEnc(K_d, IV_d , k)
26:     v ← service.mapGet(Key, k')
27:     if v ≠⊥ then
28:         return rndDec(K_r, IV_r , v)
29:     return ⊥
30:
31: function GetAll( Key )
32:     encState ← service.mapGetAll(Key)
33:     state ← {}
34:     for (k, v) ∈ encState do
35:         k' ← detDec(K_d, IV_d , k)
36:         v' ← rndDec(K_r, IV_r , v)
37:         state ← state ∪{(k', v')}
38:     return state
39:
40: function Size( Key )
41:     return service.mapSize(Key)
```

Algorithm 18 Counter - Part I

```
1:  function Counter( index )
2:      i ← index
3:      p ← {}
4:      n ← {}
5:      mp ← {}                          ▷ Auxiliary structure for the p vector
6:      mn ← {}                          ▷ Auxiliary structure for the n vector
7:
8:  function Inc( delta )
9:      ts ← mp[i]
10:     p[i] ← p[i] + delta
11:     mp[i] ← generateTimestamp()
12:
13: function Dec( delta )
14:     ts ← mn[i]
15:     n[i] ← n[i] + delta
16:     mn[i] ← generateTimestamp()
17:
18: function Get( )
19:     sumP ← 0
20:     sumN ← 0
21:     for id ∈ p do
22:         sumP ← sumP + p[id]
23:     for id ∈ n do
24:         sumN ← sumN + n[id]
25:     return sumP - sumN
26:
27: function Merge( p', mp', n', mn' )
28:     for (id', ts') ∈ mp' do
29:         ts ← mp[id']
30:         if ts' > ts then
31:             p[id'] ← p'[id']
32:             mp[id'] ← mp'[id']
33:     for (id', ts') ∈ mn' do
34:         ts ← mn[id']
35:         if ts' > ts then
36:             n[id'] ← n'[id']
37:             mn[id'] ← mn'[id']
```

Algorithm 19 Counter Stub

```
1:  function RegisterStub( p, q )          ▷ p and q are paillier's private numbers
2:      pl ← Paillier(p, q);
3:      service ← getServiceEndpoint();
4:
5:  function Inc( Key, delta )
6:      delta' ← pl.Enc(delta)
7:      service.counterInc(Key, delta')
8:
9:  function Dec( Key, delta )
10:     delta' ← pl.Enc(delta)
11:     service.counterDec(Key, delta')
12:
13: function Get( Key )
14:     ctr ← service.counterGet(Key)
15:     return pl.Dec(ctr)
```

3.3.6 Security and critical analysis

Throughout the previous subsections we only discussed the encryption of the data that is stored in the structures with no mention of the metadata that is also stored, namely the time stamps and the type of operation. The main reason for this is that this information is not worth to hide or obfuscate as its very difficult to hide from an adversary which possesses access to the machines where the data store service is executing and that is monitoring them in order to deduce information. For instance, this adversary could be the cloud administrator which is an adversary we contemplate on our threat model.

When it comes to the encryption schemes that we leverage for our SCRDTs, the deterministic encryption scheme is the one more prone to reveal information about the stored data. Specifically it reveals repetitions in structures that admit repetition among their elements, such as the list, and overall whether a given encrypted element belongs or not in the structure. The first case is a direct consequence of the encryption scheme being deterministic as it will always produce the same ciphertext for the same plaintext. The second case is also a consequence of determinism but it also requires the adversary to infer information about the type of computations that are being performed. Nonetheless when the data is at rest, i.e. when no computations are being performed over it, and in structures were no repetitions are allowed, the deterministic encryption scheme is as secure as a probabilistic encryption scheme. If the data is not at rest, then the amount of information revealed increases with the amount of operations performed. A possible solution would be to periodically refresh the ciphertext by re-encrypting it with new parameters. However, if there is a large amount of data this solution is not very practical and may hinder the system's availability.

The structures that use partial homomorphic and probabilistic encryption schemes are by default more secure as these schemes present IND-CPA security levels [51].

Chapter 4

Implementation

In this chapter we discuss and present the implementation aspects of our solution. We implemented our solution incrementally in two phases. First, we developed the library that contained the data structures discussed in Section 3.3, both the secure (SCRDT) and non-secure (CRDT) version. We developed our own set of structures as we could not find an existing library of CRDTs in JAVA that featured the all the necessary structures.

Then we also developed a data store prototype which contained said structures. However, due to time constraints we could not completely consolidate the implementation of the prototype in order to experimentally evaluate it. In Section 4.1 of this chapter we present a description of said prototype.

As a programming language for both the library and prototype we used Java. For the prototype we used the Akka toolkit [1]. Akka is an implementation of the Actor Model proposed by Carl Hewitt [34] for the JVM that is geared towards the development of applications that are highly concurrent, distributed and fault-tolerant. These are common traits of a data store, hence our decision to us this toolkit for our prototype. Table 4.1 shows a summary of the technologies used.

Table 4.1: Summary of the technologies used.

Technology	**Version**
Java	8
Akka	2.5.4
Google Protocol Buffers	3

The implementation of the library was straightforward given the pseudocode we have already presented in Section 3.3, therefore we will not address it [1].

[1] The source code can be found at `https://github.com/tavares-jdst/crypto_crdts`.

In Section 4.1 we discuss how we implemented our prototype using the aforementioned Akka toolkit modules. This prototype follows the system model shown in Section 3.1.

4.1 Data store prototype

To develop our prototype, we used three modules from the Akka toolkit: Distributed Data, Cluster and HTTP. The Distributed Data module [3] offers eventually consistent, high read and write available, low latency data (i.e. CRDTs). It also enables the creation of custom distributed data and provides support for their usage and management in distributed applications. This module proved to be crucial to integrate our CRDTs and SCRDTs in our prototype.

The Cluster module [2] offers a membership service that is decentralized, fault-tolerant and peer-to-peer that uses gossip protocols for communication among the cluster's nodes and an automatic failure detector. We took advantage of this module to build, deploy and manage many replicas of the data store service in a seamless way.

The HTTP module [4], permits exporting the API of the data store service through a REST web service thus enabling clients to use it.

In the rest of this section, we explain how each of these modules were used to develop each component of our system.

Storage We developed this component using the Distributed Data module. Specifically, we integrated our structures in this module by creating custom replicated data by extending two classes — `AbstractReplicatedData` and `AbstractSerializationSupport`. The former class allowed to create the custom replicated data. The latter was required for serialization as, per the documentation guidelines, the custom data must comply with the Akka serializer. This class contains the logic to (de)serialize our structures under Google Protocol Buffers (protobuf) [31].

Due to protobuf not supporting generic types and because there was an immutability requirement for the custom replicated data, we had to adapt the implementation of our structures. The update and merge operations of each structure return a new instance, when previously they only altered the structure, to comply with the immutability requirement. Then, we altered the CRDT structures so that they store pre-defined types. In particular the map, register and set structures store strings whilst the counter stores integers. We consider this last alteration to our solution acceptable as other in memory key values such as Redis use a similar approach. For the SCRDT structures the implementation remained the same as they by default store a comparable byte array.

Replication Mechanism This component is fully supported by the `Replicator` actor offered by the Distributed Data module. Each node of the system contains an instance of this actor. Whenever a node of the system receives an operation over some structure from

the client, the `Replicator` at that node is invoked and propagates the operation to the majority of the cluster node's by communicating with their instances of the `Replicator`. If the operation is propagated to the majority of the replicas without a timeout occurring, then it is considered to be successful.

The `Replicator` exports two operations for this purpose: update and get. Updates are to be used whenever a modifying operation of a structure is invoked, in other words updates are to be used to replicate the update operations of the structures. Gets are to be used for obtaining the current state of a structure and therefore are to be used for the query operations of a structure.

Data consistency is enforced through get operations of the `Replicator`. Upon receiving the current state of the structure from a majority of the replicas, all of the received states are merged and the result of this operation is considered to be the current state of the structure.

Client Interface To provide external access to the store service, each node has a service endpoint configured which was implemented through extending the `AllDirectives` class provided by the HTTP module. This class permits configuring routes that handle specific requests. In our API we have one route for each the different structures, which in turn have sub routes to support each of the structures' operations.

Stubs This component is supported through using the `HttpRequest` and `HttpResponse` provided by the HTTP module. As previously explained, there is a Stub for each type of structure supported by our service that is responsible for transforming the client's requests and for issuing the transformed request to a given endpoint of our service through the API exported by the Client Interface. The transformation consists in encrypting the operation's arguments under the appropriate encryption scheme which is selected based on the structure it was issued upon.

Client The client component is a simple instance of an Akka actor which uses the Stubs components to interact with the data store service. To issue requests to the system, clients select at random an endpoint from a list of service endpoints available and then invoke the intended operation through the Stubs component while providing the endpoint to be used.

Chapter 5

Experimental Evaluation

In this chapter we present our experimental evaluation which consisted in performing micro benchmarks to the developed structures. The experiment is detailed in Section 5.1 and it focuses on a pure performance comparison between the CRDTs and their SCRDT counterparts to have an understanding of the impact of enhancing the structures with security.

5.1 Micro benchmarks

This benchmark focuses on purely comparing the CRDTs and SCRDTs we developed performance wise to have a first understanding in how they compare.

As such, each micro benchmark was executed in a local environment and each measured, for a given data structure, the average execution time of 1000 of each of its operations. Beforehand, each data structure was populated with 1000 input data objects. The benchmarks were performed in a machine running Ubuntu 16.04 with an Intel(R) Core(TM) i5-5200U 2.20GHz CPU and mSATA 256 GB SSD.

As input for the benchmarks we generated, from a base input string (for the sets, lists and maps structures) or number (for the counter), a random and simple data set with the required number of items to perform the operations, namely 1000. For instance, for the add operation of a set and given an input string "benchmark", 1000 variations of this string are created by simply adding a number from 1 up to 1000 to the string. All the operations for each data structure were executed over a single replica, with the exception of the merge operations where two replicas of the data structure were used.

As the benchmarks regarding the SCRDTs will include the appropriate encryption and decryption operations, which is done at the client side, a performance penalty is expected in the results when compared to the performance results of the corresponding

CRDT.

The results obtained are show on tables 5.1 to 5.5.

Table 5.1: Map micro benchmarks results in $\mu s/op$.

Operation	CRDT LWW	SCRDT LWW	CRDT AW	SCRDT AW
Put	3.97	80.48	4.43	81.12
Remove	1.2	42.2	1.1	41.95
Get	0.65	46.9	0.8	46.37
Contains	0.78	41.24	0.58	40.09
GetAll	155.92	707.42	162.69	717.57
Merge	208.87	213.17	264.16	270.76

Table 5.2: Sets micro benchmarks results in $\mu s/op$ (N/A = Non Applicable).

Operation	CRDT GSet	SCRDT GSet	CRDT LWW	SCRDT LWW	CRDT AW	SCRDT AW
Add	2.34	45.35	8.4	51.7	43.68	83.55
Remove	N/A	N/A	1.21	42.83	12.63	57.62
Contains	0.54	43.31	0.96	43.17	3.19	44.17
GetAll	1.15	348.53	0.29	349.12	196.8	530.69
Merge	58.01	183.12	141.5	185.96	409.7	470.54

Table 5.3: Lists micro benchmarks results in $\mu s/op$.

Operation	CRDT LWW	SCRDT LWW	CRDT AW	SCRDT AW
Insert	197.11	247.54	207.95	256.1
Remove	185.8	229.03	187.64	233.5
Get	44.27	91.98	46.03	94.4
Contains	21.89	63.73	21.91	62.58
GetAll	56.71	368.87	58.3	371.04
Merge	35028.22	35219.48	34848.81	34940.21

5.1.1 Critical analysis

As expected the performance of the SCRDTs is overall worse then its CRDT counterpart.

In the map structure for the all of its operations with the exception of getall, the execution time overhead of the SCRDT versions in comparison to their CRDT counterpart is proportional to the number of encryption/decryption operations required and the time these operations take. This proportion is also observable in the set and list structures.

For the getall operation in the map this observation is not applicable as, unlike the remaining operations, the decryption of the elements requires to iterate over the whole structure thus adding up the cost of searching the structures to the cost of the decryption

Table 5.4: Register micro benchmarks results in $\mu s/op$.

Operation	CRDT	SCRDT
Put	0.86	41.15
Get	0.37	48.84
Merge	0.87	1.11

Table 5.5: Counter micro benchmarks results in $\mu s/op$.

Operation	CRDT	SCRDT
Inc	2.86	724.34
Dec	2.44	728.28
Get	18.91	2159.62
Merge	8.51	8.78

of every element. Nonetheless, the search and decryption steps were optimized to be done in parallel if possible through the use of streams and lambda expressions. Hence in the best scenario these execution time of these steps is the time of decrypting one element if total parallelism is possible plus the time to iterate the structure. In the worst scenario, it takes up to N times the time of decrypting one element if no parallelism is possible, where N denotes the number of elements in the structure. This observation is also valid for the getall operations of the the set and list structures. The register and counter structures have only one query operation — get — that returns a single value, therefore for these structures the execution time overhead of the SCRDT versions in comparison to their CRDT counterpart is in fact proportional to the number of decryption operations required and the time this operation takes.

The merge operation execution time is similar for each variant of the map CRDT and SCRDT, which is to be expected as it doesn't require any sort of encryption and it depends solely on the size of the internal state of the map. When it comes to the merge operations of the remaining structures, with the exception of the set, the execution times are also very similar due to the same fact. In the set structure the difference of executions times of the merge operations between the CRDT and SCRDT versions is unusually steep and unexpected. Another noticeable aspect in the results of the merge operations, is the values for the list structure which are very high yet expectable due to collisions of elements stored at the same position in the different copies of list. For each collision, it is necessary to compute a new composed identifier which is a fairly complex task (refer to Section 3.3.2) thus it renders the merge quite slow. A final observation is that the counter structure presents the biggest execution time disparity between its CRDT and SCRDT versions. This overhead is expectable as the underlying cryptographic primitive for its SCRDT version is Pailler's encryption scheme and each operation performs sums over the ciphertext using this primitive which is a computationally heavy operation.

Although there is an accentuated performance overhead between the CRDT and SCRDT variants, the overhead resides at the client side. This means that on the server the operations over the SCRDT structures have the same amount of execution time as their CRDT counterparts and, consequently the throughput of the system where the SCRDTs integrate would not be as hindered as expected. In conclusion, the attained results are promising and show the potential benefits of the SCRDTs.

Chapter 6

Conclusion

The cloud computing paradigm is very popular and is an attractive solution to deploy services that scale, are highly available and fault tolerant such as NoSQL data stores. However with the rise of its popularity, the number of security-related incidents also increased. Most of these incidents are associated to the privacy, confidentiality and integrity of the data stored in the cloud. The attacks that lead to the incidents might not always be carried out by outside attackers that wish to gain some knowledge about the data stored in the cloud, but they might also be carried out by the cloud provider itself.

These factors hinder the user's trust in cloud-based services, especially if the data is considered to be critical as it would if the user is an health insurance company that outsources their data in the cloud. For instance NoSQL data stores provide some security mechanisms, but these are limited when compared to traditional relational databases.

In this thesis we proposed a set of secure data abstractions that enable trusted cloud computation in the context of NoSQL data stores. Moreover, these abstractions also allow for programmers to develop their own secure distributed applications. The abstractions are a new set of secure conflict-free replicated data types, SCRDTs, which are security enhanced through the use of cryptographic schemes, such as homomorphic and property-preserving schemes, that enable their operations to be made over ciphertext.

To evaluate our solution we conducted micro benchmarks to compare each secure abstraction with their non secure counterpart performance wise. The results of our experiment shows that our abstractions provide the intended security requirements with an acceptable performance overhead, showing that it has potential to be used to build solutions for trusted cloud computation.

6.1 Contributions

In summary, the main contributions of this thesis where:

- A new design of CRDTs that are security enhanced, SCRDTs, that perform their operations over ciphertext and that allow to build secure distributed applications and to perform trusted computations.
- Two libraries, one containing a set of CRDT structures and another containing the secure version of the previous set of structures, the SCRDTs;
- A paper entitled "SCRDTs: Tipos de Dados Seguros e Replicados sem Conflitos"which was accepted for the INForum 2018 conference, thus validating the contributions of our solution for the scientific community.

6.2 Future Work

As in this thesis we focused primarily on state based replicated CRDTs and SCRDT, our first proposal for future work would be to extend our solution to incorporate operation based replicated CRDTs and SCRDTs so that it becomes even more rich and complete. Our second proposal for future work is an iteration of the SCRDTs where instead of providing security guarantees through only the use of cryptographic techniques, the guarantees would be also provided through the use of trusted hardware technologies (refer to Section 2.1).

Briefly, trusted hardware technologies enable the creation of TEEs that provide confidentiality and integrity guarantees both to the input and output data and the computations that are performed within the environment. We shall focus on Intel SGX as the trusted hardware technology because as previously discussed (refer to Sections 2.1.4.1 and 2.1.5), TPM's are slow and inefficient which renders it inappropriate performance wise and when compared to ARM's Trustzone, Intel SGX it has a smaller TCB and a well defined threat model,which are desirable traits.

To combine the cryptographic technologies and TEEs, we have two possible approaches that differ in the way in which Intel SGX would be used in combination with some cryptographic technology. The first approach would consist in combining partial homomorphic and property preserving encryption with Intel SGX. Given a cloud provider which offers nodes as computation resources that run on physical machines, which may or may not enable the usage of Intel SGX, the update operations of the structures are either carried out (i) using an enclave and over the encrypted data or (ii) outside an enclave and directly over the encrypted data.

Naturally, if Intel SGX is available (this can be verified through reading a subset of CPU registers [62]) then option (i) would take place, otherwise option (ii) would take place. This approach not only allows for interoperability among both technologies, but

it permits to obfuscate to a degree the HbC cloud adversary by minimizing leakage of information upon executing computations. Specifically it allows to hide the nature of the metadata that is generated upon the invocation of the update operation. This is possible because the enclave is an environment that is completely isolated from the normal execution environment, therefore it cannot be monitored by an adversary that aims to infer this information, such as the HbC cloud administrator.

The second approach would be to combine probabilistic encryption with Intel SGX. Given a cloud provider which offers nodes as computation resources that run on physical machines which enable the usage of Intel SGX, the operations of the structures are carried out using an enclave and over unencrypted data. In this approach, the data is kept encrypted under the aforementioned scheme if it is at rest. Upon an operation over a structure, an enclave is created to execute it. Both the instruction to be executed and the structure with the encrypted data are loaded to the enclave. To execute the operation, first the data must be decrypted therefore the required cryptographic key must be provided by the client. This can be done one of two ways, either the client may send it over with each request through the TLS channel or the key may be already stored and sealed within an enclave. Then after the instruction is carried out, the data of the structure is re-encrypted before the structure is stored away. This approach, similarly to the previous one, permits to obfuscates the HbC cloud adversary by hiding the nature of the computations being performed as they are carried out within the enclave. But unlike the previous solution, that uses property preserving encryption schemes, this one hides completely patterns within the data that is kept in the structures due to the usage of the probabilistic encryption scheme.

Bibliography

[1] *Akka*. https://akka.io/. Online; accessed September 2018.

[2] *Akka Cluster*. https://doc.akka.io/docs/akka/2.5.4/java/cluster-usage.html. Online; accessed September 2018.

[3] *Akka Distributed Data*. https://doc.akka.io/docs/akka/2.5.4/java/distributed-data.html. Online; accessed September 2018.

[4] *Akka HTTP*. https://doc.akka.io/docs/akka-http/current/introduction.html. Online; accessed September 2018.

[5] *Apache Cassandra*. http://cassandra.apache.org/doc/latest/. Online; accessed 3 February 2018.

[6] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. "Orthogonal Security with Cipherbase." In: *CIDR*. 2013.

[7] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. "Transaction processing on confidential data using cipherbase." In: *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE. 2015, pp. 435–446.

[8] ARM. *Building a Secure System using TrustZone® Technology*. Tech. rep. ARM, 2009.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. "SCONE: Secure Linux Containers with Intel SGX." In: *OSDI*. 2016, pp. 689–703.

[10] N Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. "Mobile trusted computing." In: *Proceedings of the IEEE* 102.8 (2014), pp. 1189–1206.

[11] Y Benslimane, Z Yang, and B Bahli. "Key Topics in Cloud Computing Security: A Systematic Literature Review." In: *Information Science and Security (ICISS), 2015 2nd International Conference on*. IEEE. 2015, pp. 1–4.

[12] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. "DepSky: dependable and secure storage in a cloud-of-clouds." In: *ACM Transactions on Storage (TOS)* 9.4 (2013), p. 12.

[13] D. Boneh, E.-J. Goh, and K. Nissim. "Evaluating 2-DNF Formulas on Ciphertexts." In: *TCC*. Vol. 3378. Springer. 2005, pp. 325–341.

[14] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. "TOCTOU, traps, and trusted computing." In: *International Conference on Trusted Computing*. Springer. 2008, pp. 14–32.

[15] T. Brito, N. O. Duarte, and N. Santos. "ARM TrustZone for Secure Image Processing on the Cloud." In: *Reliable Distributed Systems Workshops (SRDSW), 2016 IEEE 35th Symposium on*. IEEE. 2016, pp. 37–42.

[16] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation." In: *NDSS*. Vol. 14. 2014, pp. 23–26.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A distributed storage system for structured data." In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[18] D. Chen and H. Zhao. "Data security and privacy protection issues in cloud computing." In: *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*. Vol. 1. IEEE. 2012, pp. 647–651.

[19] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. ": outsourcing computation without outsourcing control." In: *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM. 2009, pp. 85–90.

[20] V. Costan and S. Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[21] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. "Searchable symmetric encryption: improved definitions and efficient constructions." In: *Journal of Computer Security* 19.5 (2011), pp. 895–934.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[23] J.-E. Ekberg, K. Kostiainen, and N Asokan. "The untapped potential of trusted execution environments on mobile devices." In: *IEEE Security & Privacy* 12.4 (2014), pp. 29–37.

[24] T. ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms." In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[25] B. Ferreira. "Privacy-preserving efficient searchable encryption." Doctoral dissertation. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2016.

[26] B. Ferreira, B. Portela, T. Oliveira, G. Borges, H. Domingos, and J. Leitão. *BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage*. Cryptology ePrint Archive, Report 2018/588. https://eprint.iacr.org/2018/588. 2018.

[27] B. Ferreira, J. Leitao, and H. Domingos. "MuSE: Multimodal Searchable Encryption for Cloud Applications." In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2018, pp. 181–190.

[28] C. Fontaine and F. Galand. "A survey of homomorphic encryption for nonspecialists." In: *EURASIP Journal on Information Security* 2007.1 (2007), p. 013801.

[29] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. "Terra: A virtual machine-based platform for trusted computing." In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 193–206.

[30] C. Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009.

[31] *Google Protocol Buffers*. https://developers.google.com/protocol-buffers/docs/overview. Online; accessed September 2018.

[32] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz. "Data management in cloud environments: NoSQL and NewSQL data stores." In: *Journal of Cloud Computing: advances, systems and applications* 2.1 (2013), p. 22.

[33] R. Hecht and S. Jablonski. "NoSQL evaluation: A use case oriented survey." In: *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE. 2011, pp. 336–341.

[34] C. Hewitt. "Viewing control structures as patterns of passing messages." In: *Artificial intelligence* 8.3 (1977), pp. 323–364.

[35] *Intel Software Guard Extensions SDK*. https://software.intel.com/en-us/documentation/sgx-sdk-developer-reference. Online; accessed 15 January 2018.

[36] M. S. Islam, M. Kuzu, and M. Kantarcioglu. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation." In: *Ndss*. Vol. 20. 2012, p. 12.

[37] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment." In: *NDSS*. 2015.

[38] K. Kursawe, D. Schellekens, and B. Preneel. "Analyzing trusted platform communication." In: *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware*. 2005.

[39] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. "Search pattern leakage in searchable encryption: Attacks and new construction." In: *Information Sciences* 265 (2014), pp. 176–188.

[40] R. Macedo, J. Paulo, R. Pontes, B. Portela, T. Oliveira, M. Matos, and R. Oliveira. "A Practical Framework for Privacy-Preserving NoSQL Databases." In: *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*. IEEE. 2017, pp. 11–20.

[41] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA* 10 (2013).

[42] P. Mell, T. Grance, et al. "The NIST definition of cloud computing." In: (2011).

[43] M. A. Mohamed, O. G. Altrafi, and M. O. Ismail. "Relational vs. nosql databases: A survey." In: *International Journal of Computer and Information Technology* 3.03 (2014), pp. 598–601.

[44] *MongoDB Manual - Security*. https://docs.mongodb.com/manual/security/. Online; accessed 3 February 2018.

[45] M. Naveed. "The Fallacy of Composition of Oblivious RAM and Searchable Encryption." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 668.

[46] *Neo4j Documentation*. https://neo4j.com/docs/. Online; accessed 3 February 2018.

[47] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. "Oblivious Multi-Party Machine Learning on Trusted Processors." In: *USENIX Security Symposium*. 2016, pp. 619–636.

[48] P. Paillier and D. Pointcheval. "Efficient public-key cryptosystems provably secure against active adversaries." In: *Asiacrypt*. Vol. 99. Springer. 1999, pp. 165–179.

[49] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. "Big Data Analytics over Encrypted Datasets with Seabed." In: *OSDI*. 2016, pp. 587–602.

[50] N. C. Paxton. "Cloud security: a review of current issues and proposed solutions." In: *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*. IEEE. 2016, pp. 452–455.

[51] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. "CryptDB: Processing Queries on an Encrypted Database." In: *Commun. ACM* 55.9 (Sept. 2012), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/2330667.2330691. URL: http://doi.acm.org/10.1145/2330667.2330691.

[52] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. "A commutative replicated data type for cooperative editing." In: *ICDCS*. IEEE. 2009, pp. 395–403.

[53] *Redis Documentation*. https://redis.io/documentation. Online; accessed 3 February 2018.

[54] *Riak KV*. http://basho.com/products/riak-kv/. Online; accessed 3 February 2018.

[55] R. L. Rivest, L. Adleman, and M. L. Dertouzos. "On data banks and privacy homomorphisms." In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[56] N. Santos, K. P. Gummadi, and R. Rodrigues. "Towards Trusted Cloud Computing." In: *HotCloud* 9.9 (2009), p. 3.

[57] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services." In: *USENIX security symposium*. 2012, pp. 175–188.

[58] N. Santos, H. Raj, S. Saroiu, and A. Wolman. "Using ARM TrustZone to build a trusted language runtime for mobile applications." In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM. 2014, pp. 67–80.

[59] S. Savvides, J. J. Stephen, M. S. Ardekani, V. Sundaram, and P. Eugster. "Secure data types: a simple abstraction for confidentiality-preserving data analytics." In: *Proceedings of the 2017 Symposium on Cloud Computing*. ACM. 2017, pp. 479–492.

[60] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. "VC3: Trustworthy data analytics in the cloud using SGX." In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 38–54.

[61] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. "SGX-Shield: Enabling address space layout randomization for SGX programs." In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2017.

[62] *SGX-Hardware*. https://github.com/ayeks/SGX-hardware. Online; accessed 16 February 2018.

[63] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "A comprehensive study of convergent and commutative replicated data types." Doctoral dissertation. Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[64] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[65] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs." In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2017.

[66] E. R. Sparks and E. R. Sparks. "A security assessment of trusted platform modules computer science technical report TR2007-597." In: *Dept. Comput. Sci., Dartmouth College, Hanover, NH, USA, Tech. Rep., TR2007-597* (2007).

[67] W. Stallings and L. Brown. *Computer Security: Principles and Practice*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 0133773922, 9780133773927.

[68] *TPM Library Specification*. https://trustedcomputinggroup.org/tpm-library-specification/. Online; accessed 22 January 2018.

[69] L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, and A. V. Vasilakos. "Security and privacy for storage and computation in cloud computing." In: *Information Sciences* 258 (2014), pp. 371–386.

[70] *What is MongoDB?* https://www.mongodb.com/what-is-mongodb. Online; accessed 3 February 2018.